Quantum Advantage for All

CHRISTOPH M. KIRSCH, University of Salzburg, Austria and Czech Technical University, Prague, Czech Republic
STEFANIE MUROYA LEI, Czech Technical University, Prague, Czech Republic

We show that the algorithmic complexity of any classical algorithm written in a Turing-complete programming language polynomially bounds the number of quantum bits that are required to run and even symbolically execute the algorithm on a quantum computer. In particular, we show that any classical algorithm A that runs in $O(f(n))$ time and $O(g(n))$ space requires no more than $O(f(n) \cdot g(n))$ quantum bits to execute, even symbolically, on a quantum computer. With $O(1) \leq O(g(n)) \leq O(f(n))$ for all n, the quantum bits required to execute A may therefore not exceed $O(f(n)^2)$ and may come down to $O(f(n))$ if memory consumption by A is bounded by a constant. Our construction works by encoding symbolic execution of machine code in a finite state machine over the satisfiability-modulo-theory (SMT) of bitvectors, for modeling CPU registers, and arrays of bitvectors, for modeling main memory. The FSM is linear in the size of the code, independent of execution time and space, and represents the reachable machine states for any given input. The FSM may be explored by bounded model checkers using SMT and SAT solvers as backend. However, for the purpose of this paper, we focus on quantum computing by unrolling and bit-blasting the FSM into (1) satisfiability-preserving quadratic unconstrained binary optimization (QUBO) models targeting adiabatic forms of quantum computing such as quantum annealing, and (2) semantics-preserving quantum circuits (QCs) targeting gate-model quantum computers. Given a bound on n, both QUBOs and QCs are generated in $O(f(n) \cdot g(n))$ time and space. From that we derive two major insights: (1) through our QUBOs and in particular their relatively compact size, real quantum annealers can now execute simple but real code even symbolically, yet only with potential but no guarantee for exponential speedup, and (2) through our QCs used as oracles, Grover's algorithm applies to symbolic execution of arbitrary code, guaranteeing at least in theory a quadratic speedup of symbolic execution on gate-model quantum computers.

1 INTRODUCTION

We have designed and implemented *unicorn*[1][2], an open-source toolchain that enables symbolic execution of machine code through bounded model checking as well as adiabatic and gate-model quantum computing. Unicorn consists of three components: the *BEATOR* frontend generating finite state machines for symbolic execution through bounded model checking (Section 3) as well as the *QUBOT* backend generating quadratic unconstrained binary optimization (QUBO) models for adiabatic quantum computing (Section 4) and the *QUARC* backend generating quantum circuits (QCs) for gate-model quantum computing (Section 5).

Given a program P written in a Turing-complete subset of C, BEATOR encodes symbolic execution of 64-bit and 32-bit RISC-V machine code generated from P in a finite state machine over the theory of bitvectors for modeling CPU registers and arrays of bitvectors for modeling main memory. The FSM represents the reachable machine states for any given input. Each transition in the FSM corresponds to executing one machine instruction (or reading one machine word as input). Which instruction is determined by a Boolean flag called pc flag, one for each instruction in the code. Without any modifications to the code, BEATOR scans the code in a linear fashion two times, first generating combinational circuits for register and memory updates (data flow), and then generating combinational circuits for pc flag updates (control flow).

BEATOR is fast, generating the FSM in $\Theta(|P|)$ time and space. Optionally, given a bound m on memory size, BEATOR may generate an equivalent FSM over just the theory of bitvectors without arrays of bitvectors in $\Theta(m \cdot |P|)$ time and space by unfolding all main memory access into pure register access. For this purpose, BEATOR generates upon each memory access a combinational circuit that switches over all m memory addresses to return the correct memory entry.

[1]Original prototype in C and Python (discontinued): https://github.com/cksystemsteaching/selfie/tree/unicorn
[2]Standalone prototype in Rust (in development as of 2022): https://github.com/cksystemsgroup/unicorn

BEATOR outputs FSMs in a file format called BTOR2 [40] for which a bounded model checker called `btormc` [40] exists that was instrumental in debugging and validating BEATOR.

Given a machine state M and a bound n on execution steps, QUBOT unrolls the FSM generated by BEATOR n times, propagates constants, and bit-blasts the result into a combinational circuit that is encoded in a satisfiability-preserving QUBO model B such that any solution of B contains input on which P runs into M executing no more than n instructions, and conversely, if B has no solution, there is no input on which P runs into M executing no more than n instructions.

QUBOT is also fast, generating the model in $O(n \cdot (n \cdot |P|))$ time and space where n also bounds memory size since each machine instruction may only expand the machine state by a constant amount of bits. Moreover, if memory consumption of P is bounded by a constant, QUBOT's time and space complexity comes down to $O(n \cdot |P|)$. QUBOT also supports unfolding main memory access. Doing that in QUBOT rather than BEATOR may in practice result in QUBO models with fewer quantum bits. QUBOT stores QUBO models in our custom-designed format for which we developed an auxiliary tool for debugging and validating QUBOT by evaluating QUBO models on known inputs. We also implemented support in QUBOT for deploying QUBO models on real quantum annealers from D-Wave systems. Section 6 has the details.

QUARC does essentially the same as QUBOT, even with slightly better time and space complexity if considering machine word size, but generates a semantics-preserving quantum circuit Q, rather than a QUBO model, such that Q outputs 1 for input on which P runs into M executing no more than n instructions, and 0 otherwise. Unlike QUBOT, QUARC relies on BEATOR for unfolding main memory access. QUARC outputs OpenQASM [15], an open standard for specifying quantum circuits with tool support for validation, simulation, and deployment on real gate-model quantum computers.

Unicorn improves the state of the art in at least three dimensions with the following contributions:

(1) (Software) Unicorn is the first tool that translates software written in a Turing-complete programming language that supports unbounded dynamic stack and heap allocation (procedures, malloc). Unicorn targets both adiabatic and gate-model quantum computing. The state of the art in that dimension is an algorithm that translates a subset of C that resembles a hardware description language that is not Turing-complete and does not support dynamic memory allocation [41]. Moroever, that algorithm does not support symbolic execution and bounded model checking and only targets quantum annealers.
(2) (Space) Unicorn generates the to-date asymptotically most compact models, which are smaller by a linear factor in execution steps over the state of the art which is, to the best of our knowledge, a tool called CheckFence [11] for finding concurrency bugs. CheckFence transforms code into register SSA form prior to translation into models related to our FSMs. Unicorn avoids register SSA form by generating circuits that switch over all machine instructions that update a given register.
(3) (Time) Unicorn may eventually speed up symbolic execution (1) in practice by generating compact QUBO models that can already be solved by existing quantum annealers, and (2) in theory by targeting gate-model quantum computing and thus guaranteeing quadratic speedup in finding inputs through Grover's algorithm [26].

Unicorn's improvements over existing work [11, 41] are not achieved by mere extension or refinement. Instead, unicorn and in particular BEATOR are based on a novel, uniform encoding of control and data flow as well as dynamic memory access developed from scratch which suppports symbolic execution of arbitrary code through bounded model checking, and quantum computing using QUBOT and QUARC as backend.

While all of unicorn is available as open source, its complexity is significant making both its presentation and proof of correctness difficult. The latter remains future work that is likely going to involve proof assistants and automation. For the presentation of unicorn we focus on algorithmic complexity of all relevant components and explain algorithmic details by example using the same running example throughout the paper. Detailed pseudo code of representative versions of BEATOR, QUBOT, and QUARC can be found in supplementary material.

Quantum Programming Models

The background on quantum computing necessary to follow this paper requires understanding the logics but not the physics of the quantum programming models that are relevant here. As previously mentioned, we distinguish adiabatic quantum computing, in particular quantum annealers from gate-model quantum computers. Quantum annealers tend to have at least one order of magnitude more quantum bits (qubits) than existing gate-model machines, and are therefore an interesting, working target for unicorn using QUBOT as backend. In Section 6, we report on our running example actually executing symbolically on a real quantum annealer by D-Wave systems. Gate-model machines are also an interesting target for unicorn with QUARC as backend but only for obtaining theoretical results, at least for now. Unfortunately, we have no access to a real gate-model machine.

The programming models of quantum annealers and gate-model machines that we use here are QUBO models and quantum circuits in OpenQASM [15], respectively. A QUBO model is a binary quadratic function for which we would like to find a solution, that is, an assignment of its binary variables such that the function evaluates to zero, or else know that there is no solution. The problem of solving QUBO models is NP-hard. A quantum annealer, on the other hand, features a number of qubits where each qubit has a programmable bias towards 0 or 1. Moreover, each qubit can be programmed to be entangled with another qubit. Entanglement is a phenomenon in quantum mechanics that allow two qubits to correlate or anticorrelate their value. Both, bias and entanglement are programmable by normalized real values. A quantum annealer can solve a QUBO model by associating each binary variable x of the model with a unique qubit q on the annealer. Here, the terms (binary) variable and qubit can therefore be seen as logically synonymous. The linear factor of x in the model is represented by the bias of q on the annealer. A bi-linear[3] factor of x with another variable y in the model is represented by the entanglement of q and the qubit associated with y on the annealer.

Once linear and bi-linear factors are configured as bias and entanglement, respectively, quantum annealing refers to the process of moving from a high energy state in which all qubits are in superposition to a low (ground) energy state in which all qubits are 0 or 1 and represent a solution of the QUBO model with some probability that can be increased by repeating the process and tuning the model. The time to find out if there is a solution or not and, if there is, find a solution, depends on the model. In particular, the time is inversely proportional to the minimum gap of the model, which is the difference in energy between the ground state and the first excited state of the model [21]. In short, the closer non-solutions are to the ground state the longer it takes to find an actual solution. Another issue is that qubits can usually not be entangled with all other qubits. This can be addressed by representing a single binary variable with multiple physical qubits known as ancillae. Thus annealing QUBO models also involves solving an NP-hard problem called *minor embedding* that searches for the least number of ancillae needed to represent a QUBO. However, minor embeddings can be solved efficiently and sufficiently accurate using heuristics. The following related work section and Section 6 have more on that.

[3] an example of a bi-linear factor is $-2xy$ whereas a quadratic factor such as $-2x^2$ is a linear factor if x is a binary variable because then $-2x^2 = -2x$

Understanding quantum circuits and gate-model machines is more involved than understanding QUBO models and quantum annealers. However, since with QUARC we only translate combinational circuits to quantum circuits, we only need to know that any combinational circuit can be translated to a semantics-preserving quantum circuit of same asymptotic size. A gate-model machine can then be programmed to solve another NP-hard problem: finding inputs to a quantum circuit that make the circuit output, say, 1. Here, the time to do so may in the worst case be exponential in the number of input bits but, without any further information on the circuit, can be reduced at least by a quadratic factor using Grover's algorithm [26]. Independently of that, the key challenge in QUARC as well as in QUBOT is to reduce the need for qubits so that the generated QUBO models and quantum circuits eventually fit on real quantum hardware.

2 RELATED WORK

We found inspiration from two, separate lines of thought that we bring together here: (1) translation of classical code to quantum annealers [27, 41] and (2) symbolic execution [30], from systems such as KLEE [12] and S2E [14], and from bounded model checking [8], such as the bounded model checker `btormc` [40]. Satisfiability Modulo Theory (SMT) solvers such as Z3 [19] and boolector [39], and the fact that SAT and SMT formulae can be solved using quantum annealers have helped us as well [7, 9, 20, 47, 55].

While BEATOR and QUBOT together logically resemble translation of Verilog to QMASM via EDIF [41], there are a number of technical advancements as well as an important principled difference: firstly, unicorn translates a Turing-complete subset of C, in particular code using dynamic memory, and not Verilog [41] or domain-specific C code targeting quantum annealers [27]. Secondly, unlike [41], it supports symbolic execution and bounded model checking not just on the theory of bitvectors but, most importantly, on the theory of arrays of bitvectors as well which is key to supporting dynamic memory allocation. Thirdly, it enables relating classical algorithmic complexity with spatial quantum complexity by lifting translation of classical code [41] to symbolic execution and bounded model checking on quantum computers. However, unicorn currently defers handling minor embeddings to a D-Wave library[4] at the expense of improved annealing performance [41].

BEATOR improves asymptotic model size over CheckFence [11] by a linear factor making models generated by BEATOR linear in code size. Unlike CheckFence, BEATOR avoids transforming code into register SSA form and instead translates code directly to combinational circuits that switch over all instructions that update a given register. However, which choice provides better performance in practice is unclear.

BTOR2 [40] extends SMT-LIB [4] with sequential operators that inspired us in how to avoid the path and state explosion problems when encoding symbolic execution of arbitrary code. Translating BTOR2 to quadratic unconstrained binary optimization (QUBO) models takes us to the domain of QUBO problems and solvers. QUBO is NP-hard [18] and has numerous applications beyond quantum annealing [32]. QUBO problems can also be solved by algorithms that target gate-model quantum computers [23, 38, 50, 57], and other classical algorithms such as Steepest Gradient Descent [36] and Tabu Search [22] as well as variations of Tabu Search [6, 24, 37, 43, 60], Simulated Quantum Annealing (SQA) [16, 52], Global Equilibrium Search (GES) [45, 46]. SQA algorithms use Markov Chain Monte Carlo methods [16] and classical accelerators such as GPUs and FPGAs [52] to deal with the problem that current quantum computers feature only relatively small amounts of qubits. Optimization techniques found in QUBO solvers but also SMT solvers and bounded model checkers as well as in compilers may eventually help reducing the number of qubits when translating RISC-V via BTOR2 to QUBO.

[4] www.dwavesys.com

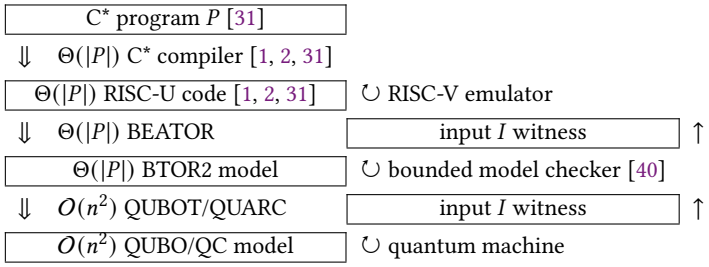

Fig. 1. Unicorn toolchain and workflow

Moving on to quantum computing, there are essentially two mainstream models of quantum computers or Noisy Intermediate-Scale Quantum (NISQ) devices: quantum annealing (QA) and gate-model (GM) machines. Our results directly apply to both, QA using QUBOT and GM using QUARC, while both are also at least in principle interchangable [28, 56]. In the NISQ-devices market [17], QA greatly outperforms GM in number of available qubits with the most recently released Quantum Processor Unit (QPU) featuring 5k qubits, and the announcement of a QPU with 7k qubits. Moreover, QA has recently been shown to return global optima with higher probability than GM while both, QA and GM, do not provide fairness if multiple ground states exist [44]. QA has already been applied in various domains other than executing arbitrary code [3, 35, 48, 51, 53].

In principle, QA can solve any QUBO model. However, for QA to work QUBO models still need to be minor-embedded onto the particular architecture of a machine where not all qubits can be entangled with all other qubits. Finding proper minor embeddings [5, 10, 18, 59] is an NP-hard problem itself and important to enable quantum annealing and increase performance [41]. Improving the QUBO models to enhance the solution space [34, 42, 49, 58], considering how various parameters in the annealing process affect results [25, 29], and addressing the problem of too many binary variables versus too few available qubits [33] is important as well.

3 CLASSICAL MODELING WITH BEATOR

Fig. 1 provides an overview of the unicorn toolchain. We first introduce, by running example, the part of the toolchain that translates a Turing-complete subset of C called C* [31] via a subset of RISC-V [54] called RISC-U [2] to a subset of BTOR2 [40]. The compiler from C* to RISC-U has been implemented in C* independently of unicorn and is used here as is [1, 2, 31]. BEATOR, the RISC-U-to-BTOR2 translator we introduce here, is written in C* in around 3.5-KLOC utilizing the compiler as frontend. Program inputs found during symbolic execution can be validated by any RISC-V emulator including a RISC-U emulator that is part of the compiler [1, 2, 31]. BTOR2 models can be validated by the bounded model checker `btormc` [40].

3.1 C* [31]

Consider the source code of the running example in Fig. 2. The program reads a single byte from the console keyboard and returns zero as exit code unless the user presses `1` in which case a segmentation fault may be triggered by the attempt to read from unallocated memory with `*(x + a)` since `a` is then `'0'` which is `48`, and not `0`. Below we consider certain types of unsafe memory access triggering segmentation faults as well as non-zero exit codes and division by zero as machine states to look for. However, the toolchain is able to compute program input that leads to any given

```
uint64_t* x;
uint64_t main() { uint64_t a;
  x = malloc(1); // rounded up to 8
  // touch to trigger page fault here
  *x = 0;
  // read 1 byte from console into x
  read(0, x, 1);
  // copy from heap to stack segment
  a = *x;
  // decrement input until <= '0'
  while (a > '0')
    a = a - 1;
  // segmentation fault on input '1'
  if (a == *x - 1) // '0' == '1' - 1
    // segfault: '0' != 0
    a = *(x + a);
  return 0;
}
```

Fig. 2. Running example of a C* program

machine state. Program input is all `read` input. Once some input has been determined the code can be executed on that input to validate if a machine state is actually reached using, for example, the RISC-U emulator.

The code is written in C* which is a tiny subset of C that has originally been developed for educational purposes [1, 2, 31]. The example essentially features all elements of C* except procedure calls. C* only features two data types, `uint64_t` and `uint64_t*`, and five statements: assignment, `while`, `if`, `return`, and procedure call. There are the usual arithmetic and comparison operators but only for unsigned 64-bit integer arithmetic. Notably, there is only the unary dereference operator `*` to access heap memory. There are no arrays and no structs hence the name C*. Furthermore, C* supports integer, character, and string literals as well as global and local variables and procedure parameters. Lastly, there is `printf` library support and a total of five builtin procedures: `exit`, `open`, `read`, `write`, and `malloc`. Turns out that, because of its overall simplicity and in particular its focus on unsigned integer arithmetic, C* is well-suited as target for researching and prototyping tools for symbolic execution [12, 19, 30, 40].

The part of the toolchain that is relevant here is written in C* and consists of a non-optimizing linear-time (modulo global (local) symbol table hash collisions (search)) C* compiler that targets RISC-U, a tiny subset of 64-bit RISC-V [54], as well as a RISC-U emulator and BEATOR which translates RISC-U code to BTOR2 [40]. RISC-U binaries generated by the C* compiler are in ELF format and run not only on the RISC-U emulator but also on QEMU and actual 64-bit RISC-V hardware. The toolchain also fully supports 32-bit machines and generates 32-bit code by bootstrapping `uint64_t` to `uint32_t` and carefully avoiding integer overflows beyond 32 bits. Support of 32-bit binaries highlights the impact of machine word size on number of qubits in Section 6. Thus everything below also applies to 32-bit machines and code.

3.2 RISC-U ISA [1, 2, 31]

Consider the RISC-U assembly fragment in Fig. 3 which has been generated by the C* compiler for the `while` loop as well as the `if` and the `return` statements in the running example. The assembly fragment features 9 out of the 14 RISC-U instructions. Overall code size is linear in the size of the C* source. Labels and comments are provided manually.

```
WHILE: ld  t0,-8(s0)         //t0=a
       addi t1,zero,48       //t1='0'
       sltu t0,t1,t0         //t0='0'<a
       beq  t0,zero,6[IF]    //goto IF '0'=>a
       ld   t0,-8(s0)         //t0=a
       addi t1,zero,1         //t1=1
       sub  t0,t0,t1          //t0=a-1
       sd   t0,-8(s0)         //a=a-1
       jal  zero,-8[WHILE]   //goto WHILE
IF:    ld   t0,-8(s0)        //t0=a
       ld   t1,-16(gp)       //t1=x
       ld   t1,0(t1)         //t1=*x
       addi t2,zero,1        //t2=1
       sub  t1,t1,t2         //t1=*x-1
       sub  t0,t1,t0         //t0=*x-1-a
       addi t1,zero,1        //t1=1
       sltu t0,t0,t1         //t0=*x-1-a<1
       beq  t0,zero,8[R0]    //goto R0 *x-1!=a
A0:    ld   t0,-16(gp)       //t0=x
       ld   t1,-8(s0)        //t1=a
       addi t2,zero,8        //t2=8 (bytes)
       mul  t1,t1,t2         //t1=a*8
A1:    add  t0,t0,t1         //t0=x+a*8
SEGFL: ld   t0,0(t0)         //t0=*(x+a)
       sd   t0,-8(s0)        //a=*(x+a)
R0:    addi t0,zero,0        //t0=0
       addi a0,t0,0          //a0=0
       jal  zero,1[EXIT]     //return 0 (a0)
```

Fig. 3. RISC-U assembly fragment generated for the running example in Fig. 2

In order to familiarize yourself with the instruction set, consider Table 1 which shows syntax and semantics of the 14 RISC-U instructions. Similar to C*, RISC-U only supports unsigned 64-bit integer arithmetic, hence the name RISC-U.

A RISC-U machine has a 64-bit program counter denoted `pc`, 32 general-purpose 64-bit registers numbered `0` to `31` and denoted `zero`, `ra`, `sp`, `gp`, `tp`, `t0-t2`, `s0-s1`, `a0-a7`, `s2-s11`, `t3-t6`, and 4GB of byte-addressed main memory. Register `zero` always contains the constant value 0. RISC-U binaries only use up to 18 of the 32 general-purpose registers, namely `zero`, `ra` which stands for *return address*, `sp` for *stack pointer*, `gp` for *global pointer*, `t0-t6` where the `t` stands for *temporary*, `s0` where the `s` stands for *saved*, and `a0-a3` and `a6-a7` where `a` stands for *argument*.

RISC-U instructions are encoded in 32 bits (4 bytes) each and stored next to each other in memory such that there are two instructions per 64-bit double word. Memory, however, can only be accessed at 64-bit double-word granularity. The `d` in `ld` and `sd` stands for double word. Below we nevertheless refer to double word by machine or memory word or just word for brevity. The parameters `rd`, `rs1`, and `rs2` denote any of the general-purpose registers. The parameter `imm` denotes an *immediate* value which is a signed integer in two's complement represented by a fixed number of bits depending on the instruction. The five builtin procedures of C* are implemented by five system calls that follow the ABI of Linux and the official RISC-V toolchain enabling RISC-U binaries to run on both platforms as well.

Initialization	
`lui rd,imm`	$rd \leftarrow imm * 2^{12}; pc \leftarrow pc + 4$
`addi rd,rs1,imm`	$rd \leftarrow rs1 + imm; pc \leftarrow pc + 4$
Memory	
`ld rd,imm(rs1)`	$rd \leftarrow mem[rs1 + imm]; pc \leftarrow pc + 4$
`sd rs2,imm(rs1)`	$mem[rs1 + imm] \leftarrow rs2; pc \leftarrow pc + 4$
Arithmetic	
`add rd,rs1,rs2`	$rd \leftarrow rs1 + rs2; pc \leftarrow pc + 4$
`sub rd,rs1,rs2`	$rd \leftarrow rs1 - rs2; pc \leftarrow pc + 4$
`mul rd,rs1,rs2`	$rd \leftarrow rs1 * rs2; pc \leftarrow pc + 4$
`divu rd,rs1,rs2`	$rd \leftarrow rs1 /_{unsigned} rs2; pc \leftarrow pc + 4$
`remu rd,rs1,rs2`	$rd \leftarrow rs1 \%_{unsigned} rs2; pc \leftarrow pc + 4$
Comparison	
`sltu rd,rs1,rs2`	$rd \leftarrow \begin{cases} 1, & \text{if } rs1 <_{unsigned} rs2 \\ 0, & \text{otherwise} \end{cases}$ $pc \leftarrow pc + 4$
Control	
`beq rs1,rs2,imm`	$pc \leftarrow \begin{cases} pc + imm, & \text{if } rs1 == rs2 \\ pc + 4, & \text{otherwise} \end{cases}$
`jal rd,imm`	$rd \leftarrow pc + 4; pc \leftarrow pc + imm$
`jalr rd,imm(rs1)`	$tmp \leftarrow ((rs1 + imm)/_{unsigned} 2) * 2;$ $rd \leftarrow pc + 4; pc \leftarrow tmp$
System	
`ecall`	system call number in $a7$, parameters in $a0$–$a3$, return value in $a0$

Table 1. Table of the 14 RISC-U instructions [31]

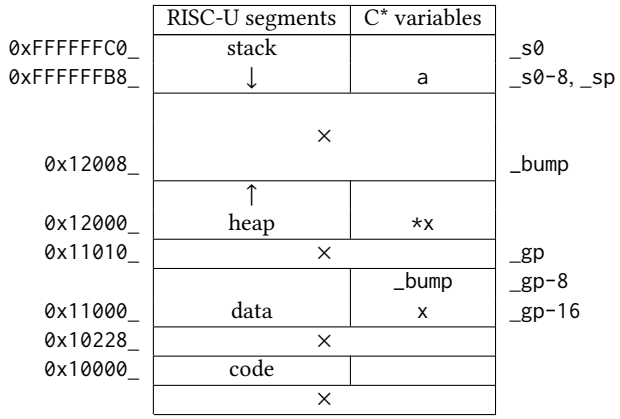

Fig. 4. 4KB-page-aligned RISC-U memory segments and machine state when executing the RISC-U code in Fig. 3

The RISC-U memory layout is shown in Fig. 4. Code starts by RISC-V convention at `0x10000` and ends for our running example at `0x10228`. The data segment starts 4KB-page-aligned after the code segment at `0x11000` and ends at `0x11010`, again for our running example. The data segment contains values of global variables, string literals, and big integer literals that require more than 32 bits up to 64 bits. The layout of the data segment does not change during

code execution. Entries are addressed relative to register gp which is initialized by the executed code, here to 0x11010, and then never changed. In our running example, the data segment contains two 64-bit memory words, one for the global variable x at gp-16 and one for an internal variable _bump at gp-8 which facilitates a bump-pointer allocator for malloc. The _bump pointer is initialized by the executed code to the start of the heap which is 4KB-page-aligned after the end of the data segment, here at 0x12000. After executing x = malloc(1);, the _bump pointer is increased by 8 bytes (64 bits), rounded up from 1 byte, to 0x12008. The allocated address returned by malloc is the previous value of _bump which is 0x12000, hence making x point to the first 64-bit memory word *x on the heap. The byte read from the console keyboard by the read(0, x, 1) call is stored in the least significant byte of that memory word with the more significant bytes set to 0.

The stack segment starts at the address where register sp points to, here 0xFFFFFFB8, which is the top of the call stack. The end of the stack segment is the highest address in main memory. Thus the call stack grows downwards to lower addresses while the heap grows upwards to higher addresses towards the stack. There is currently no free call in the system but it could be added without affecting modeling. The call stack and register sp are initialized by the boot loader. Local variables and procedure parameters on the call stack are accessed relative to register s0 called the *frame pointer*. Here, the local variable a is at the top of the stack at s0-8 where s0 is set to 0xFFFFFFC0 by the callee of the currently executed procedure, that is, here the main procedure.

In sum, the state of a RISC-U machine consists of the 64-bit program counter, the 18 general-purpose 64-bit registers, and the 64-bit memory words in the data, heap, and stack segments. BEATOR as well as QUBOT and QUARC must handle that state.

3.3 BTOR2 [40] (BEATOR2)

BTOR2 is a formalism [40] for modeling finite state machines over the theory of bitvectors and arrays of bitvectors. Essentially, BTOR2 extends the bitvector fragment of SMT-LIB [4] with sequential operators for states and state transitions. BTOR2 models are input to bounded model checkers [8] such as btormc [40]. Given a BTOR2 model, a bounded model checker determines whether there is input to the model such that it transitions from its initial state to a bad state in a given number of state transitions where bad is a BTOR2 operator that essentially specifies negated safety properties [40].

The unicorn toolchain includes a RISC-U-to-BTOR2 translator called *BEATOR* that generates models in a subset of BTOR2 called BEATOR2, as summarized in Table 2 including its usage and informal relation to RISC-U. Below, we nevertheless speak of BTOR2 rather than BEATOR2 despite the ambiguity. While BEATOR supports the bad operator, support of BTOR2 operators for specifying (global) fairness constraints and (negations of) liveness properties [40] is future work. BEATOR runs in time linear and generates models in space linear in the size of RISC-U code, and thus in the size of C* programs:

Proposition 1. *Let P be a C* program, M a machine state, and n a bound on number of executed machine instructions. Then, for all program input I, the RISC-U code R generated from P runs on I into M executing no more than n machine instructions if and only if the* bad *state modeling M in the BTOR2 model generated by BEATOR for R is reachable on I in no more than $n + \lfloor |I|/w \rfloor$ state transitions where w is machine word size (reading input in RISC-U only takes a single* ecall *instruction whereas in the BTOR2 model reading input takes as many transitions as there are machine words in I).*

Examples of machine states are return of non-zero exit codes, division by zero, and segmentation faults, that is, any, possibly input-dependent memory access outside of the data, heap, and stack segments. We added checking for

BEATOR2 ($\subset$ BTOR2)	Usage and Relation to RISC-U
`sort bitvec` *size*	value types (64-bit machine word and 1-bit Boolean)
`sort array` *sort sort*	memory type (64-bit elements, 64-bit address space)
Combinational Operators (related to SMT-LIB [4])	
`constd` *sort integer*	initial values, static addresses, immediate values
`add` *sort x y*	models `add rd,`x`,`y
`sub` *sort x y*	models `sub rd,`x`,`y
`mul` *sort x y*	models `mul rd,`x`,`y
`udiv` *sort x y*	models `divu rd,`x`,`y
`urem` *sort x y*	models `remu rd,`x`,`y
`ult` *sort x y*	models `sltu rd,`x`,`y
`uext` *sort x b*	unsigned extension of `ult`, `input` to 64-bit words
`slice` *sort x u l*	address down-scaling $u - l + 1$ bits from bit l to bit u
`ite` *sort bool x y*	`ite` cascades over pc flags for state updates
`and` *sort bool bool*	control flow of `beq` and `jalr`
`not` *sort bool*	`jalr`, $false$ branch of `beq`
`eq` *sort x y*	`jalr`, *true* branch of `beq`
`+` `neq`, `ulte`, `ugt(e)`	bound checks, kernel model
`read` *sort mem idx*	models `ld rd,`*idx*
`write` *sort mem idx val*	models `sd` *val*`,`*idx*
Sequential Operators (not in SMT-LIB [4])	
`state` *sort name*	variable of *sort* with *name*
`init` *sort state val*	initial value *val* of *state*
`next` *sort state val*	transition of *state* to *val*
`input` *sort*	symbolic input variable used in `READ` syscall model
`bad` *bool*	*bool* describes machine state

Table 2. Table of all BEATOR2 declarations and combinational and sequential operators from BTOR2 [40]

segmentation faults to BEATOR to support QUBO modeling of memory access. Optionally, BEATOR can also generate additional checks such that memory accesses outside of memory blocks allocated by `malloc` are machine states to look for. However, those checks increase the size of the BTOR2 model by around 50%.

The key idea of BEATOR is to separate control and data flow in RISC-U code as follows. For data flow, BEATOR generates zeroed 64-bit bitvectors, one for each general-purpose register, and an array of zeroed 64-bit bitvectors, one for each 64-bit memory word, indexed by a 64-bit bitvector as memory address. The array models RISC-U main memory where the data and stack segments are initialized exactly as a RISC-U bootloader initializes them prior to code execution. This also includes the stack pointer which is the only register that must be initialized by the bootloader. BEATOR supports a down-scaled linear address space and a segmenting MMU and RAM model that avoids arrays of bitvectors. QUBOT also models MMU and RAM which allows us to tradeoff translation complexity, as described in the next section and in Section 6.

For control flow, BEATOR generates zeroed 1-bit bitvectors, called pc flags, one for each instruction in the RISC-U code, for modeling the program counter. The pc flag for the instruction at the entry point of the code is the only pc flag initialized to 1 indicating that this instruction is the first to execute. From then on, control flows through the model by resetting the pc flag of the current instruction, after executing it, and then setting the pc flag of the next instruction to execute. Thus the invariant here is that at all times all pc flags are `0` except one. The alternative to pc flags is to

```
1 sort bitvec 1  ; Boolean
2 sort bitvec 64 ; 64-bit machine word
3 sort array 2 2 ; 64-bit physical memory
10 zero 1 ...
//... register states 200-231 ...
200 zero 2 zero // register $0 is always 0 ...
203 state 2 gp ; register $3 ...
205 state 2 t0 ; register $5
206 state 2 t1 ; register $6
//... program counter states ...
16603600 state 1       // beq t0,zero,8[R0]:
16603601 init 1 16603600 10
16604000 state 1    // A0:ld t0,-16(gp) ...
16606800 state 1    // R0:addi t0,zero,0
//... 64-bit memory (data,heap,stack):
20000000 state 3 physical-memory
  loading data,heap,stack into memory:
20000001 init 3 20000000 17380002
//... data flow ...    A0:ld t0,-16(gp):
36604000 constd 2 -16
36604001 add 2 203 36604000
36604003 read 2 20000000 36604001
36604004 ite 2 16604000 36604003 36603202
//...                  A1:add t0,t0,t1:
36605600 add 2 205 206
36605601 ite 2 16605600 36605600 36604004
//...               SEGFL:ld t0,0(t0):
36606002 ite 2 16606000 36606001 36605601
//...                  R0:addi t0,zero,0:
36606800 ite 2 16606800 200 36606002
//... updating registers ...
60000005 next 2 205 36606800 t0
```

Fig. 5. Data-flow fragment of the BTOR2 model for the running example

represent the program counter explicitly by a 64-bit bitvector (32 bits are actually enough here but require scaling) and then control execution of instructions by comparing that bitvector with their constant addresses in memory. The advantage is that we only need one fixed-size bitvector instead of as many pc flags as there are instructions. However, model size still remains linear in code size and model checking performance may be negatively affected too. Exploring that alternative, also in QUBOT, remains future work.

Consider Fig. 5 which shows a data-flow fragment of the BTOR2 model generated from the running example in Fig. 2. In BTOR2, comments are single-line comments that begin with a semicolon. Comments that begin with a double slash are not BTOR2 format and only inserted by us by hand for documentation. Each line begins with a node (line) identifier (nid) which must be larger than any previous nid. After the nid there is a keyword that identifies a BTOR2 operator followed by its arguments which may only be nids and integer literals. Forward references to larger nids than the current nid are not allowed.

```
11 one 1
//... data flow ...
36603600 eq 1 205 200    // $t0==$zero
36603601 not 1 36603600 // $t0!=$zero
//... control flow ...
//              beq t0,zero,8[R0]:
56603600 next 1 16603600 16603200
//           A0:ld t0,-16(gp):
56604000 and 1 16603600 36603601
56604001 next 1 16604000 56604000
// ...          sd t0,-8(s0):
56606400 next 1 16606400 16606000
// ...       R0:addi t0,zero,0:
56606800 and 1 16603600 36603600
56606801 ite 1 56606800 11 16606400
56606802 next 1 16606800 56606801
```

Fig. 6. Conditional control-flow fragment of the BTOR2 model for the running example

The model begins with sort (type) declarations of pc flags and memory word bitvectors as well as the memory array in nids 1-3. Sorts are applied strictly by `btormc` which turns out to be helpful for debugging BEATOR. The `2 2` in line 3 `sort array 2 2` refer to nid 2 as the index and element sorts of the array. Line `10 zero 1` declares the constant `0` of sort 1-bit bitvector. Lines `200-231` declare `state` variables for the 32 registers `0-31`. Next are `zero`ed `state` declarations of the pc flags where each nid begins with digit `1`, followed by the address of the instruction in decimal at runtime, followed by `00` and `01` for declaration and `init`ialization, respectively. For example, the line at nid `16603600` declares the pc flag for `beq t0,zero,8[R0]` which is stored in memory at `0x101F4` or `66036` in decimal. Memory is declared in `20000000 state 3 physical-memory` and initialized in `20000001 init 3 20000000 17380002` where nid `17380002` refers to the initial state of memory (not shown).

Then, there are `ite` (if-then-else) cascades that encode per-instruction data flow where each nid begins with digit `3`, followed by an `ite` expression that either selects the data flow of the given instruction, if its pc flag is set, or else refers to the data flow of the closest previous instruction that updates the same state variable. For example, `36606800 ite 2 16606800 200 36606002` either selects the value of register `zero` (nid `200`) for updating the value of register `t0`, if `R0:addi t0,zero,0` is currently executing (nid `16606800`), or else refers to the `ite` expression for `SEGFL:ld t0,0(t0)` at `36606002` which may also update `t0`, and so on. Finally, the `next` value of registers such as `t0` at nid `205` is determined by lines whose nids begin with digit `6` such as `60000005 next 2 205 36606800 t0` which refers to the head of the `ite` cascade for `t0` at `36606800`. Actual computation can be seen in `36605600 add 2 205 206` which adds the values of registers `t0` and `t1` (nid `206`) as instructed by `A1:add t0,t0,t1`. Line `36604003 read 2 20000000 36604001` models the memory `read` at address `gp-16` as instructed by `A0:ld t0,-16(gp)`.

Fig. 6 shows a conditional control-flow fragment of the BTOR2 model generated from the running example in Fig. 2. For example, line `56606802 next 1 16606800 56606801` sets the pc flag of `R0:addi t0,zero,0` (nid `16606800`) either if `beq t0,zero,8[R0]` is currently executing (its pc flag at nid `16603600` is set) and the value of register `t0` is equal to 0 (line at nid `36603600`), or else if `sd t0,-8(s0)` is currently executing (pc flag at nid `16606400` and the line at nid `56606801`). If `beq t0,zero,8[R0]` is currently executing and the value of register `t0` is not equal to 0 (line at

```
20 zero 2 ...
22 constd 2 2 ...
//... 1-byte input
71 sort bitvec 8 ; 1 byte ...
81 input 71       ; 1 byte ...
91 uext 2 81 56 // extending input to 64 bits
//... register states ...
202 state 2 sp ; register $2 ...
210 state 2 a0 ; register $10
211 state 2 a1 ; register $11
//... read system call ...
42000001 ite 2 42000000 211 36609200 ...
42000007 eq 1 42000006 22 // inc == 2
42000008 ite 2 42000007 92 91 ...
42000019 eq 1 42000006 28 // inc == 8
42000020 ite 2 42000019 98 42000018
42000021 add 2 211 210 ; $a1 + $a0
//  memory[$a1 + $a0] = input:
42000022 write 3 20000000 42000021 42000020
//... brk system call:
45000001 state 2 brk-bump-pointer
//... updating physical memory:
70000000 next 3 20000000 42000028
//... address >= current end of heap:
80000006 ugte 1 44000001 45000001
// address < current start of stack:
80000007 ult 1 44000001 202
80000008 and 1 80000006 80000007
// access between heap and stack:
80000009 bad 80000008 b2
```

Fig. 7. System call and bad state fragment of the BTOR2 model for the running example

nid 36603601) then the pc flag of `A0:ld t0,-16(gp)` is set (line at 56604001). Note that only the translation of `beq` as well as `jal` and `jalr` instructions results in BTOR2 code that *connects* control and data flow. While there are only finitely many jump targets with `jal` and `jalr` instructions (RISC-U binaries are static), `beq` instructions remain as the only source of path explosion with `read` system calls being the only source of data explosion, as shown next.

Lastly, Fig. 7 shows a system call and `bad` state fragment of the BTOR2 model generated for the running example. In particular, it shows how input flows into the model through a `read` system call, how a potential segmentation fault is detected as `bad` state, and how main memory is written to, in this case, through the `read` system call. One-byte input (nid 81) is unsigned-extended to a 64-bit memory word (nid 91) and then flows via an `ite` cascade (head at nid 42000020) to a `write` operator (nid 42000022). The address for the `write` operator is `a1 + a0` (nid 42000021) where register `a0` (nid 210) is a cursor over the write buffer that was originally passed to the `read` system call in register `a1` (nid 211). A potential segmentation fault, such as through `SEGFL:ld t0,0(t0)`, is detected in line `80000009 bad 80000008 b2` if there is any memory access at an address above the heap (bump pointer value of the `brk` system call at nid 45000001) and below the stack (pointer nid 202) where nid 44000001 is the head of an `ite` cascade over all

addresses used in `read` and `write` operators. There are similar `bad` states for other unsafe parts of memory. All of main memory is updated by a single `next` (nid `70000000`) that refers to the head of an `ite` cascade at `42000028` over all `write` operators in the model.

There is a fun fact that we like to mention: running BEATOR is fast and since it is written in C* it can actually model itself and the rest of the unicorn toolchain written in C* as well. The BTOR2 model of the whole C* toolchain takes less than a second to build and is around 4MB.

3.4 BEATOR Loves QUBOT Loves BEATOR

Dead code elimination, constant propagation, and *bounded memory modeling* are all effective translation techniques in reducing the number of qubits in a QUBO model. Moreover, they can be done by BEATOR or by QUBOT. We have therefore added support of all three to BEATOR. By bounded memory modeling we mean modeling a segmenting MMU and RAM in a bounded number of 64-bit bitvectors, one for each memory word in RAM, resulting in BTOR2 models that do not contain any arrays of bitvectors and thus any `read` and `write` operators anymore. The segmenting MMU bounds the size of the data, heap, and stack segments and then maps them, using the `slice` operator of BTOR2, from the 64-bit RISC-U virtual address space to a minimal physical address space. For the running example, a 4-bit physical address space is sufficient with only 12 memory words actually being accessed: 2 data words, 1 heap word, and 9 stack words while only the heap word (`*x`) and 1 stack word (`a`) are ever updated, see again Fig. 4 for the memory layout.

The RAM option takes the MMU option further by mapping each access to a memory word onto the 64-bit RAM bitvectors. RAM removes the need for arrays of bitvectors but increases BEATOR's complexity to $\Theta(m \cdot |P|)$ where m is the size of memory accessed by P. However, with RAM modeling already done in BEATOR, QUBOT's complexity effectively remains the same, maintaining the toolchain's complexity. The MMU option can be used without the RAM option and only reduces memory size without changing BEATOR's complexity. We have also implemented a third, combined MMURAM option in BEATOR that mimics QUBOT's implementation which maps virtual addresses directly to the 64-bit RAM bitvectors.

Constant propagation in BEATOR runs in $O(n)$ time by executing the first n RISC-U instructions of a binary until reaching the first `read` call. Then, BEATOR eliminates dead code and snapshots the machine state including the heap as initial state in the BTOR2 model. QUBOT takes advantage of the additional $O(n)$ time, maintaining the toolchain's complexity, and propagates constants beyond the first `read` call. However, propagating constants until the first `read` call is faster in BEATOR. Quantitative details are in Section 6.

4 QUANTUM MODELING WITH QUBOT

We introduce, by continuing to use the running example, the BTOR2-to-QUBO translator called *QUBOT* written in Python in around 3-KLOC. A QUBO model is a binary quadratic (BQ) function, which QUBOT generates from a BTOR2 model. A BQ function is a quadratic function from binary variables to positive real values and 0 (ground energy on a quantum annealer). Constant factors are signed real values that are eventually normalized before solving the function. For example, the following BQ function encodes the logic gate for $NOT(x) = y$:

$$NOT_{BQ}(x, y) = 2 - 2x - 2y + 4xy$$

Try evaluating the function to see when it reaches ground energy! Instead of generating a file, QUBOT represents a QUBO model as upper-diagonal matrix of constant factors in memory using a Python library by D-Wave Systems[4]. The diagonal of the matrix represents linear factors such as $-2x$ and the upper part represents bi-linear factors such as $4xy$.

BEATOR2 (⊂ BTOR2)	#binary variables or #qubits	
	QUBOT	QUARC
`sort bitvec, array`	0	
Combinational Operators (related to SMT-LIB [4])		
`constd`	0	
`add, sub`	$O(w)$	
`mul`	$O(w^2)$	$O(w)$
`udiv, urem`	$O(w^2)$ constrained circuit	-
`uext`	0	$O(w)$
`slice`	0	
`ite, and, not, eq`	$O(w)$	
`neq, ult(e), ugt(e)`	$O(w)$	
`read, write`	$O(m \cdot w)$	-
Sequential Operators (not in SMT-LIB [4])		
`state, input`	$O(w)$ variables	
`init`	0	
`next`	$O(m \cdot w^2 \cdot \|P\|)$	$O(m \cdot w \cdot \|P\|)$
`bad`	$O(w \cdot \|P\|)$ circuit	

Table 3. Size of QUBO model and quantum circuit components (in number of binary variables or qubits) with respect to BEATOR2 declarations and operators where $|P|$ is original C* program size, if applicable, w is machine word size, and m is memory size ($|P|$ and w are runtime constants)

Quadratic factors of binary variables such as $-2x^2$ are actually linear factors[5] because $-2x^2 = -2x$. The model may be output in various formats including visualizations, see Fig. 8. QUBOT runs in $O(n^2)$ time and space such that:

Proposition 2. *Let B be the BTOR2 model generated by the C* compiler and BEATOR for a given C* program and machine state M. Moreover, let n be a bound on number of state transitions. Then, for all model input I, the* `bad` *state in B representing M is reachable on I in no more than n state transitions if and only if the QUBO model generated by QUBOT for B and n evaluates to 0 on I.*

Thus, with more qubits on a quantum annealer than variables in the QUBO model, quantum annealing the model may reach 0 (ground) energy for all model input I that is part of the ground state if and only if the `bad` state is reachable in B on I in no more than n state transitions. If there are not enough qubits on the machine, annealing may still work using a hybrid solver [33]. The size of the model comes down to $O(n)$ if memory consumption of P is bounded by a constant.

The key challenge in the translation is to encode reachability in a state machine (BTOR2 model) in a stateless BQ function (QUBO model). Before learning how QUBOT translates our running example, let us focus on the fact that any combinational circuit can be represented by a BQ function:

Proposition 3. *Let x, y, and z be binary variables and let f be a function (logic gate) that maps x and y to z. There exists at least one BQ function g over x, y, and z such that $g(x, y, z) = 0$ if and only if $f(x, y) = z$ for all x, y, and z in $\{0, 1\}$.*

For example, for all x, y, and z in $\{0, 1\}$, $AND(x, y) = z$ if and only if $6z + 2xy - 4xz - 4yz = 0$. Even though AND and NOT are universal, we also use dedicated BQ functions for $NAND$, OR, and $AND(NOT(x), y)$ (inhibition) for

[5] thus BQ functions are actually binary bi-linear functions

encoding the combinational operators in BTOR2. Similar to SMT solvers such as Z3 [19] and boolector [39], QUBOT uses (polynomial) bit blasting except for `udiv` and `urem` which are reduced to constraints over multiplication. However, since solving those constraints is expensive, QUBOT cannot efficiently validate QUBO models generated from `udiv` and `urem`. Bit blasting both operators is future work. Memory access through `read` and `write` is bit-blasted, see below for more details. Note that composing a function from multiple BQ functions that generally share variables is straightforward using addition since the composed function is again a BQ function from binary variables to positive values and 0. Table 3 provides a summary of the size of the BQ functions for all BTOR2 operators in terms of the size w of a machine word (here 64) and the size m of memory. The impact of the size $|P|$ of the original C* program P is also mentioned, see below for more on that.

The key idea of QUBOT is to model, given a bound n, the input I and state variables S in a BTOR2 model in $O(n)$ qubits (binary variables) that are biased and entangled with another set of $O(n)$ qubits that represent the output of the combinational circuits for control and data flow on I and S. At least logically that model is then duplicated n times where, for all $0 \leq i < n$, duplicate D_i is entangled with duplicate D_{i+1} such that each qubit o in D_i that represents an output of a combinational circuit in the BTOR2 model is used in D_{i+1} as qubit that represents the part of the state variable in the BTOR2 model which o updates [41]. Input variables are introduced in each duplicate as uninitialized state variables. After that, constants and initial values of initialized state variables are propagated at bit level (not word level) through the model.

This is the logic. In reality, QUBOT builds the model incrementally propagating constants as soon as possible to keep the model from growing unnecessarily. Instead of duplicating D_i, the model for D_{i+1} is built from scratch, one combinational circuit per `next` operator at a time, immediately followed by propagating constants through that circuit. However, there is an issue: a qubit o that represents an output of a circuit must not be replaced by a constant before all its future entanglements (uses) are known and have been modeled, which in the worst case may happen at $i = n - 1$. The reason is that o might represent part of a state variable not just in D_{i+2} but also later until that part of the state variable is updated again by a qubit other than o. In short, state is memory (registers), and memory that is not updated keeps its value. Thus QUBOT remembers each such qubit o and its possibly known constant value until it can be applied safely.

Incremental constant propagation in QUBOT is highly effective—all state variables including all pc flags are initialized—but it is certainly not exhaustive. Static analysis techniques other than constant propagation for further reducing the number of qubits may apply as well but remain future work. One such opportunity in the running example is the subtraction in `a = a - 1` in the body of the `while` loop. If the loop condition `a > '0'` was known at the time of subtraction, the bit resolution of the data flow through variable `a` could remain at 8 bits rather than 64 bits because `a - 1` never overflows into the 56 more significant bits. However, through mere constant propagation QUBOT is currently unable to perform that reasoning. Nevertheless, note that constant propagation enables QUBOT to validate models on known inputs efficiently, see Section 6.

Let us now go through the running example from QUBOT's perspective. Given a BTOR2 model such as the model for our running example, QUBOT only looks for `next` and `bad` operators. For each line with a `next` operator, such as `60000005 next 2 205 36606800 t0` for data flow into registers in Fig. 5 and `70000000 next 3 20000000 42000028` for data flow into memory in Fig. 7, and `56606802 next 1 16606800 56606801` for control flow in Fig. 6, QUBOT builds the BQ functions for the referenced (here, 64-bit and 1-bit bitvector) combinational circuits recursively, here nids `36606800`, `42000028`, and `56606801`, respectively. Lines with `state`, `input`, and `constd` operators terminate the recursion. When building D_0, QUBOT also looks for lines with `init` operators that initialize the encountered `state` variables and then, together with the encountered `constd` constants, performs constant propagation as described above.

For each line with a bad operator, such as `80000009 bad 80000008 b2` for detecting potential segmentation faults in Fig. 7, QUBOT again builds the BQ function for the referenced (1-bitvector) combinational circuit recursively, here nid `80000008`, and then constrains the BQ function in an *OR* circuit over all BQ functions generated for lines with bad operators to 1 (true).

Updating memory through the line `70000000 next 3 20000000 42000028` in Fig. 7 works just like any other line with a `next` operator, except that the referenced (array) combinational circuit involves `read` and `write` operators. The issue with these operators is that QUBOT generates a BQ function that compares the referenced (64-bit bitvector) address, such as nid `42000021` in the line `42000022 write 3 20000000 42000021 42000020`, with all addresses of the entire memory to identify the correct memory word, which is then read or written, here with the output of the referenced (64-bit bitvector) combinational circuit (nid `42000020`).

In order to address scalability, QUBOT maps virtual addresses to a minimal physical address space at translation time, similar to the combined MMURAM option in BEATOR. Here, BEATOR generating bad states for detecting segmentation faults in the BTOR2 model frees QUBOT from checking again. For the running example, a 4-bit physical address space for 12 memory words is sufficient, see again Fig. 4 for the memory layout. Most importantly, constant propagation in QUBOT reduces BQ functions modeling memory access via constant addresses to BQ functions modeling register access, effectively making their size independent of memory size. While we discussed QUBOT's perspective on a BTOR2 model generated by the unicorn toolchain, we should mention that QUBOT is able to handle any model within the BEATOR2 subset of BTOR2 and thus enables bounded model checking in general within the theory of bitvectors and arrays of bitvectors on a quantum annealer.

In sum, given a C* program P and a bound n, the unicorn toolchain takes $O(n^2)$ time to generate a QUBO model for P where generating each D_i for $0 \leq i < n$ takes $O(n)$ time. Translation time reduces to $O(n)$ if memory consumption of P is bounded by a constant. Note that the size $|P|$ of the C* program P as well as the size w of a machine word are additional linear and quadratic factors, respectively, in QUBOT's time and space complexity which we nevertheless ignore here since $|P|$ and w are both runtime constants.

5 QUANTUM MODELING WITH QUARC

We introduce the BTOR2-to-OpenQASM translator called *QUARC* written in Python in around 1.5-KLOC. OpenQASM [15] is an open standard for specifying quantum circuits with tool support for validation, simulation, and deployment on real gate-model quantum computers. QUARC generates quantum circuits using X (logic not), CNOT (controlled not) gates[6], and a generalization of CNOT gates that can have an arbitrary number of control qubits known as Toffoli gates. QUARC processes BTOR2 models with the same time and space complexity as QUBOT applying the same recursion that begins at `next` and bad operators and terminates at `state`, `input`, and `constd` operators:

Proposition 4. *Let B be the BTOR2 model generated by the C* compiler and BEATOR for a given C* program and machine state M. Moreover, let n be a bound on number of state transitions. Then, for all model input I, the* bad *state in B representing M is reachable on I in no more than n state transitions if and only if the quantum circuit generated by QUARC for B and n outputs 1 on I.*

Quantum circuits generated by QUARC can be used as oracles with Grover's algorithm, a quantum search algorithm that provides quadratic speedup over brute force search in terms of the number of elements the search space has [26].

[6]The inputs of a CNOT gate are a control and a target qubit. If the control qubit is 1, then the target qubit is flipped.

In our case, the search space is given by the size of a circuit's input space since non-input qubits have either a classical value or are transitively entangled with input qubits:

Proposition 5. *Let C be a quantum circuit generated by QUARC for a given C* program, a bound n on number of state transitions, and a machine state M. If the program reads up to i bits of input executing no more than n machine instructions, then finding such input where C outputs 1, that is, the program reaches M takes $O(\sqrt{2^i}n^2)$ time using C as oracle of size $O(n^2)$ with Grover's algorithm [26].*

Even though QUBO models generated by QUBOT work in principle as oracles with Grover's algorithm as well, generating dedicated quantum circuits is more efficient in the number of qubits since quantum operators can be used to avoid full bit-blasting and reduce ancillae (qubits for storing intermediate calculations). For example, QUARC uses fewer qubits by a linear factor on word size than QUBOT for implementing multiplication by reusing qubits in reversible parts of the circuit, see Table 3. Support of division and remainder is future work. Word extension can also be done without introducing new qubits through constant propagation but remains future work as well. For now, QUARC applies constant propagation to prune branches of `ite` operators, and to reduce the number of qubits involved in multiqubit gates since they are harder to implement and take longer to execute on real hardware. The main principled contribution of this paper is:

Proposition 6. *The algorithmic time (space) complexity of an algorithm in classical computing is a linear (quadratic) upper bound on quantum space in number of qubits for running as well as symbolically executing the algorithm on quantum annealers and gate-model quantum computers.*

6 EXPERIMENTS

First of all, we managed to run real 64-bit and 32-bit C code on D-Wave's Advantage 5k-qubit quantum annealer successfully! This means that the quantum annealer was able to determine input that actually leads to bad states. In fact, the answers returned by the quantum annealer that achieved ground energy had the same value for *all* binary variables as the ones our debugging tool produce for the specific inputs returned by the quantum annealer. While we had to replace the code below the assignment `a = *x;` in the running example with just one assignment `a = *(x + a);`, which may trigger a segmentation fault for all inputs other than `0`, we still feel that this is a breakthrough and, to the best of our knowledge, the first time C code, using dynamic memory allocation (as simple as it might be) and not targeting quantum annealers [27], ran on a quantum computer through a fully automated toolchain.

Unicorn includes tools for debugging and validation of both, QUBO models generated by QUBOT and quantum circuits generated by QUARC. Debugging and validation works by propagating known input values through models and circuits to validate the output. Moreover, the values of qubits that represent bad states reveal which bad states actually occur. We used 32-bit versions of the C* programs listed in Table 4 to debug and validate QUBO models and quantum circuits. For this purpose, BEATOR was configured to perform constant propagation until first input, down-scale the linear address space, and encode MMU and RAM access, to support both QUBOT and QUARC. Additionally, QUBO models in 32- and even 64-bit versions listed in Table 8 were validated with all BEATOR configurations. Note that here validation is feasible because all programs have only a 1-byte input and thus only require checking 256 values each.

Table 4 also mentions the number of qubits that we would have needed in order to execute each of the programs symbolically. Here, QUBOT is effective in reducing the number of qubits for programs with very few state transitions because of its eager constant propagation. However, QUARC does a better job when constants cannot be further propagated since it needs, in terms of word size, even asymptotically fewer ancillae to represent each operator.

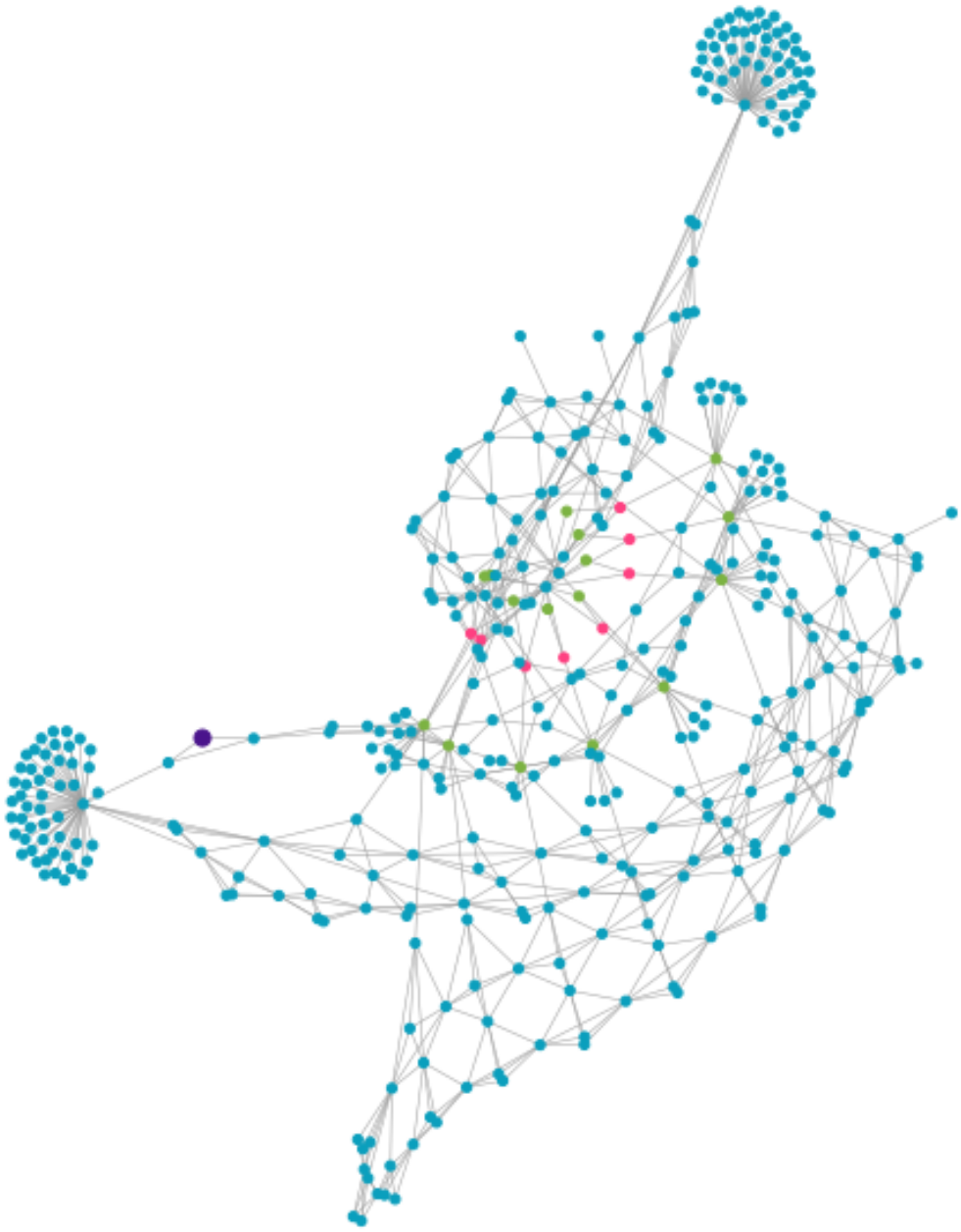

Fig. 8. QUBO model of the running example with 15 state transitions (input qubits are in pink, registers in green and pink, the big purple dot is a bad state).

program	#state transitions	bad states	#logic qubits (32-bit)	
			QUBOT	QUARC
division-by-zero	16	division by zero	5k	-
memory-access-fail	0	memory access in between data and heap segments	0	0
nested-if-else	51	non-zero exit code	220k	164k
nested-if-else-reverse	50		208k	157k
return-from-loop	35		40k	79k
simple-assignment	31		45k	68k
simple-if-else	45		65k	100k
simple-if-else-reverse	43		72k	100k
simple-if-without-else	44		107k	111k

Table 4. C* programs in 32-bit version for validating unicorn by checking that QUBO models generated by QUBOT and quantum circuits generated by QUARC identify the same bad states, inputs, and state transitions as `btormc` [40].

QPU	release year	topology	#qubits
DW2000	2017	Chimera (C16)	2048
Advantage	2020	Pegasus (P16)	5640

Table 5. D-Wave's QPUs used in our experiments, the year they were released, the topology of the graph describing how qubits can be entangled, and the total number of qubits available.

wordsize	QPU	#binary variables	#physical qubits	time (s)
32	DW2000	348	1106	8.29
	Advantage		589	3.12
64	DW2000	398	1126	12.16
	Advantage		680	5.29

Table 6. The total number of variables needed and the time spent in seconds to find a minor embedding for the 2 latest D-Wave QPUs, given QUBO models with 348 and 398 variables.

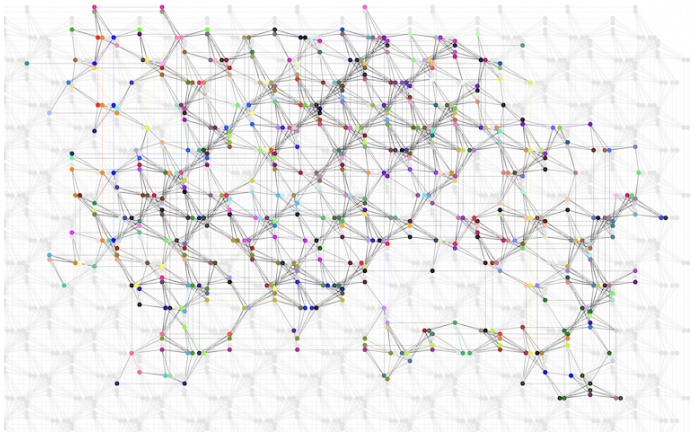

Fig. 9. Minor embedding of the modified running example in a QPU with a Pegasus topology. Nodes with the same color represent the same binary variable.

wordsize	QPU	#samples	chain strength	#minimum energy samples	minimum energy
32	P16	7k	1.5	12	0
			2	0	2
			1		6
		10k	1.5	9	0
64		7k		1	
		10k			
32	C16	7k	1.5	0	2
			2		
			2.5		4
		10k	1.5		2
64					

Table 7. Shows how varying the number of samples and chain strength affects the number of low-energy solutions found. For example, the QUBO model for 32-bit machine words achives the highest number of solutions (12) that have 0 energy when 7k samples are taken on the QPU and the chain strength is set to 1.5.

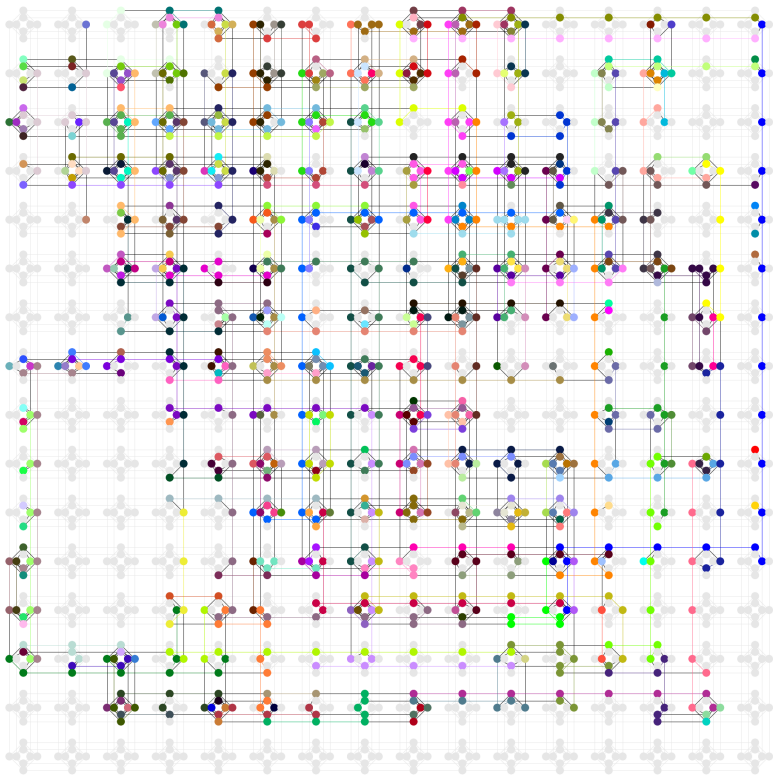

Fig. 10. Minor embedding of the modified running example in a QPU with a Chimera topology. Nodes with the same color represent the same binary variable.

We executed the running example varying only basic settings on the two latest QPUs provided by D-Wave (See Table 5). Tables 6 and 7 reveal that DW2000Q_6 fails to find solutions, doubles ancillae, and more than doubles the time

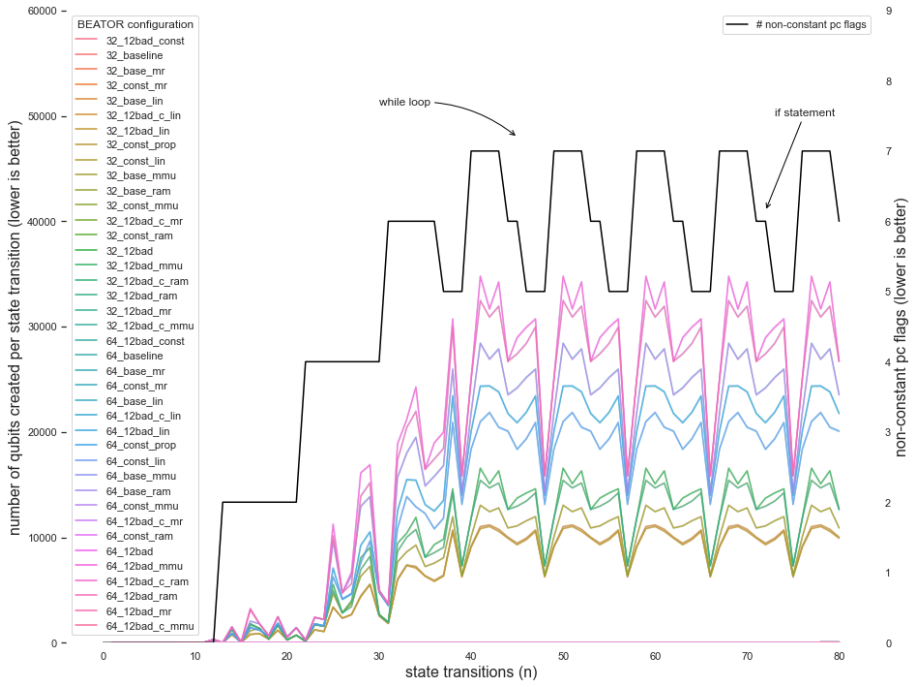

Fig. 11. Left y-axis: number of qubits per state transition (x-axis) for the running example (no growth is linear growth in total qubit count, data on 32-bit models is prefixed 32, the outliers at the top are BEATOR's RAM configurations); right y-axis: number of non-constant pc flags (number of RISC-U instructions executed simultaneously, showing (linear) impact of the `while` loop and `if` statement on qubit count)

to find a minor embedding[7] than the newer Advantage QPU. We vary the number of samples and the strength by which physical variables (chain strength) are connected to represent a logic variable. On Advantage, we got a maximum of 12 answers after sampling 7k times the 32-bit version and 1 answer for the 64-bit version. In practice, the annealing time is just 20μs per sample but additional time is needed among samples, and to program and read the QPU. Still, experiments show that in the time span between these two QPUs (3 years), there is significant improvement. Also, the more qubits and connectivity a QPU has, the easier it is to find minor embeddings. Fig. 9 shows that only a small portion of the QPU is utilized in Advantage when compared to the embedding done in DW_2000 shown in Fig. 10 for the same QUBO model.

Fig. 8 visualizes the QUBO model with 398 variables of the 64-bit version of the example executed on the quantum annealer that needs 15 state transitions to raise an exception. Vertices are the variables of the QUBO model. An edge means that there is a non-zero bi-linear factor between two variables. All state transitions may generate bad states. However, constant propagation determines a constant value for all except for one (big purple dot). This figure suggest there is a lot of potential improvement that can be done in terms of number of variables needed since only 8 variables are input (pink vertices), 24 of them belong to registers, 8 are in pink because they are inputs too while the rest (16 vertices) are in green. Input variables are reused in registers, and in main memory.

[7]We used the default algorithm by D-Wave [13].

BEATOR Config.	KB	Time (s)	#Qubits (k)
DEFAULT	51[50]	7+3[4+1]	180[92]
LINEAR	53[52]	7+3[4+1]	164[91]
MMU	78[77]	9+4[5+2]	204[106]
RAM	136[135]	9+4[5+2]	221[115]
MMURAM	100[99]	7+3[4+2]	197[100]
CONST-PROP	33[32]	**3+1[2+1]**	140[70]
CONST-LIN	34[33]	6+2**[2+1]**	126[70]
CONST-MMU	47[46]	5+2**[2+1]**	188[98]
CONST-RAM	69[68]	5+2**[2+1]**	201[105]
CONST-MR	52[51]	4+2[3+1]	173[89]
12BAD	50[49]	8+3[4+1]	180[91]
12BAD-LIN	51[50]	7+3[4+2]	163[90]
12BAD-MMU	77[75]	8+3[6+2]	204[106]
12BAD-RAM	135[133]	9+3[5+2]	220[114]
12BAD-MR	99[98]	8+3[4+2]	196[99]
12BAD-CONST	**31[30]**	**3+1[2+1]**	138[70]
12BAD-C-LIN	32[31]	**3+1[2+1]**	**125[69]**
12BAD-C-MMU	46[45]	5+3**[2+1]**	187[97]
12BAD-C-RAM	68[67]	4+2**[2+1]**	201[104]
12BAD-C-MR	51[49]	4+2**[2+1]**	172[88]

Table 8. BTOR2 model size in kilobyte (KB), QUBOT translation time in seconds (modeling time + constant-propagation time), number of generated qubits in thousands, 32-bit results in brackets (obtained with running example)

Fig. 11 shows the number of qubits created by QUBOT per state transition of the unmodified running example (Fig. 2) since it contains a more interesting control and data flow that can show interesting effects of constant propagation. The number of RISC-U instructions executed simultaneously (non-constant pc flags) shows the (linear) impact of control flow on qubit count, with the `while` loop and `if` statement showing up. Generally, the more effective BEATOR configurations result in slower qubit growth in QUBOT. Gaining quantum advantage essentially requires improving that data.

Improvements may come from both BEATOR and QUBOT. Table 8 shows, for the running example, BTOR2 model size in kilobyte (KB), QUBOT translation time in seconds (model build time + constant propagation time), number of generated qubits in thousands, and 32-bit results in brackets. We interpret the data as follows: (1) the lowest QUBOT translation time and qubit count is reached if BEATOR, in decreasing order of efficacy, propagates constants until the first input is read (CONST-PROP), down-scales 64-bit virtual addresses to 29-bit [30-bit] linear addresses (LINEAR,LIN), and only generates bad states for segmentation faults (12BAD), (2) modeling MMU and RAM is more effective if done by QUBOT (on bit level versus on word level in BEATOR (MMU,RAM,MMURAM)), and (3) 32-bit models have, as expected, roughly half the qubit count of 64-bit models.

Constant propagation until first input is read is more effective in BEATOR than in QUBOT because the resulting program state is generally smaller. However, after that, constant propagation in QUBOT is highly effective, for example: 12BAD-C-LIN for 32-bit code results in around 3 million qubits without constant propagation in QUBOT. Downscaling to smaller linear address spaces is only slightly more effective in BEATOR than constant-propagating unused addressing bits in QUBOT while reducing the number of bad states in BEATOR has almost no effect. MMU and RAM modeling

with constant propagation on bit level is generally more effective in QUBOT while support of 32-bit models can only be done in the toolchain before QUBOT.

7 CONCLUSIONS AND FUTURE WORK

We have shown how to encode symbolic execution of RISC-V machine code compiled from C code in (1) finite state machines over the theory of bitvectors and arrays of bitvectors, (2) quadratic unconstrained binary optimization (QUBO) models, and (3) quantum circuits, enabling not just bounded model checkers but also quantum annealers and gate-model machines to execute arbitrary code symbolically.

Our construction implies that the algorithmic complexity of any classical algorithm written in a Turing-complete programming language polynomially bounds the number of quantum bits that are required to run the algorithm on a quantum computer. In particular, any classical algorithm A that runs in $O(f(n))$ time and $O(g(n))$ space requires no more than $O(f(n) \cdot g(n))$ quantum bits to run on a quantum computer. With $O(1) \leq O(g(n)) \leq O(f(n))$ for all n, the quantum bits required to run A may therefore not exceed $O(f(n)^2)$ and may come down to $O(f(n))$ if memory consumption is bounded by a constant.

There are numerous directions for principled future work where outperforming, that is, gaining quantum advantage over classical symbolic execution and bounded model checking is at the top of the challenges. So far, we have dealt with the fundamental tradeoff between translation and execution complexity by putting most hard, that is, exponential work into execution. Any static analysis technique applied at translation time that scales to the size of our models and circuits is likely to produce significant progress in reducing the amount of qubits required to solve problems in practice. QUBO models and quantum circuits play a key role in programming quantum annealers and gate-model machines but may require more attention to be established as key abstraction from quantum physics, similar to the role of Boolean algebra in classical computing.

There are also numerous technical challenges of which we list just a few. Besides the obvious such as full C, RISC-V, and BTOR2 support, there is also need for support of floating-point arithmetic, more efficient handling of symbolic memory addresses, detection of properties other than safety, and more advanced system call handling that covers incorrect use of system calls.

8 ACKNOWLEDGEMENTS

We thank Armin Biere for inspiring us to build BEATOR! This work has been supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

REFERENCES

[1] A.S. Abyaneh, S. Bauer, C.M. Kirsch, P. Mayer, C. Mösl, C. Poncelet, S. Seidl, A. Sokolova, and M. Widmoser. 2018. Selfie: Towards Minimal Symbolic Execution. In *Online Proc. Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs)*.

[2] A.S. Abyaneh and C.M. Kirsch. 2021. ASE: A Value Set Decision Procedure for Symbolic Execution. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM.

[3] Steven H. Adachi and Maxwell P. Henderson. 2015. Application of Quantum Annealing to Training of Deep Neural Networks. arXiv:1510.06356 [quant-ph]

[4] C. Barrett, A. Stump, and C. Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. International Workshop on Satisfiability Modulo Theories*, A. Gupta and D. Kroening (Eds.).

[5] G. Bass, M. Henderson, J. Heath, and J. Dulny. 2021. Optimizing the Optimizer: Decomposition Techniques for Quantum Annealing. *Quantum Mach. Intell.* (2021). https://doi.org/10.1007/s42484-021-00039-9

[6] R. Battiti and M. Protasi. 1997. Reactive Search, a History-Sensitive Heuristic for MAX-SAT. *ACM J. Exp. Algorithmics* 2 (1 1997), 2–es. https://doi.org/10.1145/264216.264220

[7] Z. Bian, F. Chudak, W. Macready, A. Roy, R. Sebastiani, and S. Varotti. 2017. Solving SAT and MaxSAT with a Quantum Annealer: Foundations and a Preliminary Report. In *Frontiers of Combining Systems*, C. Dixon and M. Finger (Eds.). Springer, Cham, 153–171.

[8] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS (LNCS, Vol. 1579)*. Springer, 193–207.

[9] M.L. Bonet, J. Levy, and F. Manyà. 2007. Resolution for Max-SAT. *Artificial Intelligence* 171, 8 (2007), 606–618. https://doi.org/10.1016/j.artint.2007.03.001

[10] K. Boothby, A. D. King, and A. Roy. 2020. Fast clique minor generation in Chimera qubit connectivity graphs. arXiv:1507.04774 [cs.DM]

[11] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. *SIGPLAN Not.* 42, 6 (jun 2007), 12–21. https://doi.org/10.1145/1273442.1250737

[12] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 209–224.

[13] J. Cai, B. Macready, and A. Roy. 2014. A practical heuristic for finding graph minors. https://doi.org/10.48550/ARXIV.1406.2741

[14] V. Chipounov, V. Kuznetsov, and G. Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 265–278.

[15] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. https://doi.org/10.48550/ARXIV.1707.03429

[16] E. Crosson and A. W. Harrow. 2016. Simulated Quantum Annealing Can Be Exponentially Faster Than Classical Simulated Annealing. In *Proc. Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 714–723. https://doi.org/10.1109/FOCS.2016.81

[17] E. J. Crosson and D. A. Lidar. 2021. Prospects for Quantum Enhancement with Diabatic Quantum Annealing. *Nature Reviews Physics* 3, 7 (5 2021), 466–489. https://doi.org/10.1038/s42254-021-00313-6

[18] P. Date, R. Patton, C. Schuman, and T. Potok. 2019. Efficiently Embedding QUBO Problems on Adiabatic Quantum Computers. *Quantum Information Processing* 18 (03 2019). https://doi.org/10.1007/s11128-019-2236-3

[19] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340.

[20] A. Douglass, A. D. King., and J. Raymond. 2015. Constructing SAT Filters with a Quantum Annealer. In *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver (Eds.). Springer, Cham, 104–120.

[21] Edward Farhi, Jeffrey Goldstone, Sam Sam Gutmann, and Michael Sipser. 2000. Quantum Computation by Adiabatic Evolution. https://doi.org/10.48550/ARXIV.QUANT-PH/0001106

[22] M. Gendreau and J. Potvin. 2005. *Tabu Search.* Springer, Boston, MA, 165–186. https://doi.org/10.1007/0-387-28356-0_6

[23] Austin Gilliam, Stefan Woerner, and Constantin Gonciulea. 2021. Grover Adaptive Search for Constrained Polynomial Binary Optimization. *Quantum* 5 (apr 2021), 428. https://doi.org/10.22331/q-2021-04-08-428

[24] F. Glover, G. Kochenberger, and B. Alidaee. 1998. Adaptive Memory Tabu Search for Binary Quadratic Programs. *Management Science* 44 (03 1998), 336–345. https://doi.org/10.1287/mnsc.44.3.336

[25] E. Grant, T. S. Humble, and B. Stump. 2021. Benchmarking Quantum Annealing Controls with Portfolio Optimization. *Physical Review Applied* 15, 1 (1 2021). https://doi.org/10.1103/physrevapplied.15.014012

[26] L.K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proc. Symposium on Theory of Computing (STOC)* (Philadelphia, Pennsylvania, USA). ACM, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[27] M.W. Hassan, S. Pakin, and W. Feng. 2019. C to D-Wave: A High-level C Compilation Framework for Quantum Annealers. In *High Performance Extreme Computing Conference (HPEC)*. IEEE.

[28] H. Khetawat, A. Atrey, G. Li, F. Mueller, and S. Pakin. 2019. Implementing NChooseK on IBM Q Quantum Computer Systems. In *Reversible Computation*, M. K. Thomsen and M. Soeken (Eds.). Springer International Publishing, 209–223. https://doi.org/10.1007/978-3-030-21500-2_13

[29] A. D. King and C. C. McGeoch. 2014. Algorithm engineering for a quantum annealing platform. arXiv:1410.2628 [cs.DS]

[30] J.C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[31] C.M. Kirsch. 2017. Selfie and the Basics. In *Proc. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM.

[32] G. Kochenberger, J. Hao, F. Glover, M. Lewis, Z. Lü, H. Wang, and Y. Wang. 2014. The Unconstrained Binary Quadratic Programming Problem: A Survey. *Journal of Combinatorial Optimization* 28 (07 2014). https://doi.org/10.1007/s10878-014-9734-0

[33] K. Kurowski, J. Weglarz, M. Subocz, R. Rózycki, and G. Waligóra. 2020. Hybrid Quantum Annealing Heuristic Method for Solving Job Shop Scheduling Problem. In *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. Sloot, S. Brissos, and J. Teixeira (Eds.). Springer, Cham, 502–515.

[34] M. Lewis and F. Glover. 2017. Quadratic Unconstrained Binary Optimization Problem Preprocessing: Theory and Empirical Analysis. arXiv:1705.09844 [cs.AI]

[35] A. Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics* 2 (2014). https://doi.org/10.3389/fphy.2014.00005

[36] J. C. Meza. 2010. Steepest Descent. *WIREs Computational Statistics* 2, 6 (2010), 719–722. https://doi.org/10.1002/wics.117

[37] A. Misevicius. 2005. A Tabu Search Algorithm for the Quadratic Assignment Problem. *Computational Optimization and Applications* 30 (01 2005), 95–111. https://doi.org/10.1007/s10589-005-4562-x

[38] Giacomo Nannicini. 2019. Performance of hybrid quantum-classical variational heuristics for combinatorial optimization. *Phys. Rev. E* 99 (Jan 2019), 013304. Issue 1. https://doi.org/10.1103/PhysRevE.99.013304

[39] A. Niemetz, M. Preiner, and A. Biere. 2015. Boolector 2.0 System Description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2015), 53–58.

[40] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. 2018. Btor2, BtorMC and Boolector 3.0. In *Computer Aided Verification*, H. Chockler and G. Weissenbacher (Eds.). Springer, 587–595.

[41] S. Pakin. 2019. Targeting Classical Code to a Quantum Annealer. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3297858.3304071

[42] S. Pakin. 2021. A Simple Heuristic for Expressing a Truth Table as a Quadratic Pseudo-Boolean Function. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 218–224. https://doi.org/10.1109/QCE52317.2021.00039

[43] G. Paul. 2011. An Efficient Implementation of the Robust Tabu Search Heuristic for Sparse Quadratic Assignment Problems. *European Journal of Operational Research* 209 (03 2011), 215–218. https://doi.org/10.1016/j.ejor.2010.09.009

[44] E. Pelofske, J. Golden, A. Bärtschi, D. O'Malley, and S. Eidenbenz. 2021. Sampling on NISQ Devices: "Who's the Fairest One of All?". arXiv:2107.06468 [quant-ph]

[45] O. Shylo, D. Korenkevych, and P. Pardalos. 2012. Global Equilibrium Search Algorithms for Combinatorial Optimization Problems. 277–286. https://doi.org/10.1007/978-3-642-32964-7_28

[46] V. P. Shylo and O. V. Shylo. 2010. Solving the Maxcut Problem by the Global Equilibrium Search. *Cybernetics and Systems Analysis* 46, 5 (01 9 2010), 744–754. https://doi.org/10.1007/s10559-010-9256-4

[47] J. Su, T. Tu, and L. He. 2016. A Quantum Annealing Approach for Boolean Satisfiability Problem. In *Proc. ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2897973

[48] Z. Tabi, K. H. El-Safty, Z. Kallus, P. Haga, T. Kozsik, A. Glos, and Z. Zimboras. 2020. Quantum Optimization for the Graph Coloring Problem with Space-Efficient Embedding. *Proc. International Conference on Quantum Computing and Engineering (QCE)* (10 2020). https://doi.org/10.1109/qce49297.2020.00018

[49] R. Tanburn, E. Okada, and N. Dattani. 2015. Reducing multi-qubit interactions in adiabatic quantum computation without adding auxiliary qubits. Part 1: The "deduc-reduc" method and its application to quantum factorization of numbers. arXiv:1508.04816 [quant-ph]

[50] Takashi Tsuchimochi, Yoohee Ryo, and Seiichiro L. Ten-no. 2022. Improved algorithms of quantum imaginary time evolution for ground and excited states of molecular systems. https://doi.org/10.48550/ARXIV.2205.01983

[51] H. Ushijima-Mwesigwa, C. F. A. Negre, and S. M. Mniszewski. 2017. Graph Partitioning using Quantum Annealing on the D-Wave System. arXiv:1705.03082 [quant-ph]

[52] H. M. Waidyasooriya and M. Hariyama. 2020. A GPU-Based Quantum Annealing Simulator for Fully-Connected Ising Models Utilizing Spatial and Temporal Parallelism. *IEEE Access* 8 (2020), 67929–67939. https://doi.org/10.1109/ACCESS.2020.2985699

[53] B. Wang, F. Hu, H. Yao, and C. Wang. 2020. Prime Factorization Algorithm Based on Parameter Optimization of Ising Model. *Scientific Reports* 10 (04 2020). https://doi.org/10.1038/s41598-020-62802-5

[54] A. Waterman, Y. Lee, D.A. Patterson, and K. Asanović. 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1.* Technical Report UCB/EECS-2016-118. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[55] S. Weaver, K. Ray, V. Marek, A. Mayer, and A. Walker. 2014. Satisfiability-based Set Membership Filters. *Journal on Satisfiability, Boolean Modeling and Computation* 8 (01 2014), 129–148. https://doi.org/10.3233/SAT190095

[56] E. Wilson, F. Mueller, and S. Pakin. 2021. Mapping Constraint Problems onto Quantum Gate and Annealing Devices. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. 110–117. https://doi.org/10.1109/QCS54837.2021.00016

[57] Ehsan Zahedinejad and Arman Zaribafiyan. 2017. Combinatorial Optimization on Gate Model Quantum Computers: A Survey. https://doi.org/10.48550/ARXIV.1708.05294

[58] Z.Bian, F. Chudak, R. Israel, B. Lackey, W.G. Macready, and A. Roy. 2014. Discrete optimization using quantum annealing on sparse Ising models. *Frontiers in Physics* 2 (2014). https://doi.org/10.3389/fphy.2014.00056

[59] S. Zbinden, A. Bärtschi, H. Djidjev, and S. Eidenbenz. 2020. Embedding Algorithms for Quantum Annealers with Chimera and Pegasus Connection Topologies. In *High Performance Computing*, P. Sadayappan, B. Chamberlain, G. Juckeland, and H. Ltaief (Eds.). Springer, 187–206.

[60] Y. Zhou, J. Wang, and J. Yin. 2013. A Directional-Biased Tabu Search algorithm for Multi-Objective Unconstrained Binary Quadratic Programming Problem. *Proc. International Conference on Advanced Computational Intelligence (ICACI)* (2013), 281–286.

SUPPLEMENTARY MATERIAL

In the following, we present the BEATOR algorithm in detail and an outline of the QUBOT and QUARC algorithms to support the propositions in the paper. Formal proofs remain future work which is likely to involve proof assistants and automation.

Given a C* program P compiled to a w-bit RISC-U binary R, BEATOR translates R in $O(m \cdot |P|)$ time to a $O(m \cdot |P|)$ BTOR2 model B where w is either 64 or 32 and m is either 1, for all BEATOR configurations other than MMURAM and RAM, or a bound on the size of memory accessed by P. QUBOT translates B in $O(n \cdot m \cdot w^2 \cdot |P|)$ time to a $O(n \cdot m \cdot w^2 \cdot |P|)$ QUBO model Q where n is a bound on the number of executed RISC-U instructions in R. Since every RISC-U instruction may only increase the size of memory by at most one memory word, n and m may replace each other. Finally, QUARC translates B in $O(n \cdot m \cdot w \cdot |P|)$ time to a $O(n \cdot m \cdot w \cdot |P|)$ quantum circuit C.

The BEATOR algorithm shown below is for the 32-bit $O(m \cdot |P|)$ MMURAM configuration which models memory by a finite number of 32-bit memory words avoiding the use of the theory of bitvector arrays through `read` and `write` operators. BEATOR executes the pseudo code in the order in which it appears here. Lines that contain a BTOR2 keyword generate output. The nid of each generated line is either represented by a constant or an identifier that starts with `nid_`. Variables are printed in italics. Code that checks for errors in the input is omitted. The full BEATOR algorithm for all configurations is available in source code at https://github.com/cksystemsteaching/selfie/blob/unicorn/tools/beator.c.

The QUBOT and QUARC algorithms shown below are for BTOR2 models generated with the MMURAM or RAM configurations of BEATOR. They use bit blasting to implement all combinational BTOR2 circuits. We nevertheless only show the algorithmic complexity in number of qubits (binary variables). The full QUBOT and QUARC algorithms are available in source code at https://github.com/cksystemsteaching/selfie/tree/unicorn/tools/qubot and https://github.com/cksystemsteaching/selfie/tree/unicorn/tools/quarc, respectively.

```
1 sort bitvec 1 ; Boolean
2 sort bitvec 32 ; 32-bit machine word
6 sort bitvec physical_address_space_size_in_bits ; example 4

10 zero 1
11 one 1

20 zero 2
21 one 2
22 constd 2 2
23 constd 2 3
24 constd 2 4

; start of data segment in 32-bit virtual memory
30 constd 2 start_of_data_segment ; example: 0x11000
; end of data segment in 32-bit virtual memory
31 constd 2 end_of_data_segment ; example: 0x11008

; start of heap segment in 32-bit virtual memory (initial program break)
32 constd 2 start_of_heap_segment ; example: 0x12000
; current end of heap segment in 32-bit virtual memory (current program break)
33 constd 2 end_of_heap_segment ; example: 0x12004

; allowed end of heap segment in 32-bit virtual memory (with 0B allowance)
34 constd 2 allowed_end_of_heap_segment ; example: 0x12004
; allowed start of stack segment in 32-bit virtual memory (with 0B allowance)
35 constd 2 allowed_start_of_stack_segment ; example: 0xFFFFFFDC

; highest address in 32-bit virtual address space (4GB)
50 constd 2 4294967292 ; 0xFFFFFFFC

; kernel-mode flag
60 state 1 kernel-mode
61 init 1 60 10 kernel-mode ; initial value is false
62 not 1 60

; unsigned-extended inputs for byte-wise reading
71 sort bitvec 8 ; 1 byte
72 sort bitvec 16 ; 2 bytes
73 sort bitvec 24 ; 3 bytes

81 input 71 1-byte-input
82 input 72 2-byte-input
83 input 73 3-byte-input

91 uext 2 81 24 ; uext-1-byte-input
92 uext 2 82 16 ; uext-2-byte-input
93 uext 2 83 8 ; uext-3-byte-input
94 input 2 4-byte-input
```


Fig. 12. 32-bit BEATOR header

```
; 32 32-bit general-purpose registers
; non-zero initial register values
for (r = 1; r < 32; r++)
  if (register_value[r] != 0)
    nid_r_initial_value constd 2 register_value[r]

; registers
nid_0_register zero 2 zero ; register 0 is always 0
for (r = 1; r < 32; r++) {
  nid_r_register state 2 register_name[r]
  nid_register_flow[r] = nid_r_register;
}

; initializing registers
for (r = 1; r < 32; r++)
  if (register_value[r] != 0)
    nid_r_initialize_register init 2 nid_r_register nid_r_initial_value register_name[r]
```

Fig. 13. 32-bit BEATOR register definitions

```
; 32-bit program counter encoded in Boolean flags
for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4)
  if (pc is reachable) {
    nid_pc_flag: state 1
    if (pc == entry_point)
      nid_pc_flag_initial_value: init 1 nid_pc_flag 11 ; initial program counter
    else
      nid_pc_flag_initial_value: init 1 nid_pc_flag 10
  }
```

Fig. 14. 32-bit BEATOR pc flag definitions

```
; physical memory
; data segment
for (v = start_of_data_segment, p = 0; v <= highest_32_bit_address; v = v + 4, p = p + 1) {
  if (v == end_of_data_segment) {
    ; heap segment
    v = start_of_heap_segment;
  }
  if (v == allowed_end_of_heap_segment) {
    ; stack segment
    v = allowed_start_of_stack_segment;
  }
  nid_p_virtual_address   constd 2 v ; example: 0x11000 vaddr
  nid_p_initial_value     constd 2 memory_value[v] ; example: 0x12000 word
  nid_p_RAM_word          state 2 RAM-word-p
  nid_p_initialize_memory init 2 nid_p_RAM_word nid_p_initial_value
  nid_RAM_write_flow[p] = nid_p_RAM_word;
}
size_of_physical_memory_in_words = p
```

Fig. 15. 32-bit BEATOR physical memory

```
; data flow
nid_division_flow = 21; // division by one is ok
nid_remainder_flow = 21; // remainder by one is ok
nid_start_access_flow = 30; // access in data segment is ok
nid_ecall_flow = 10; // initially, no ecall active
for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4) {
  if (pc: lui rd,imm) {
    if (rd != zero) {
      nid_pc_df0 constd 2 imm<<12
      nid_pc_df1 ite 2 nid_pc_flag nid_pc_df0 nid_register_flow[rd]
      nid_register_flow[rd] = nid_pc_df1;
    }
  } else if (pc: addi rd,rs1,imm) {
    if (rd != zero) {
      nid_pc_df0 constd 2 imm
      nid_pc_df1 add 2 nid_rs1_register nid_pc_df0
      nid_pc_df2 ite 2 nid_pc_flag nid_pc_df1 nid_register_flow[rd]
      nid_register_flow[rd] = nid_pc_df2;
    }
  } else if (pc: [add|sub|mul|divu|remu|sltu] rd,rs1,rs2) {
    if (rd != zero) {
      if (pc: divu rd,rs1,rs2) {
        nid_pc_df0 ite 2 nid_pc_flag nid_rs2_register nid_division_flow
        nid_division_flow = nid_pc_df0;
      } else if (pc: remu rd,rs1,rs2) {
        nid_pc_df0 ite 2 nid_pc_flag nid_rs2_register nid_remainder_flow
        nid_remainder_flow = nid_pc_df0;
      }
      if (pc: sltu rd,rs1,rs2) {
        nid_pc_df0 ult 1 nid_rs1_register nid_rs2_register
        nin_pc_df1 uext 2 nid_pc_df0 31
      } else
        nid_pc_df1 [add|sub|mul|udiv|urem] 2 nid_rs1_register nid_rs2_register
      nid_pc_df2 ite 2 nid_pc_flag nid_pc_df1 nid_register_flow[rd]
      nid_register_flow[rd] = nid_pc_df2;
    }
  } else ...
```

Fig. 16. 32-bit BEATOR data flow part I

```
  ... if (pc: lw rd,rs1,imm) {
    if (rd != zero) {
      nid_pc_lw_imm constd 2 imm
      nid_pc_lw_add add 2 nid_rs1_register nid_pc_lw_imm
      nid_pc_lw_acc ite 2 nid_pc_flag nid_pc_lw_add nid_start_access_flow
      nid_start_access_flow = nid_pc_lw_acc;
      nid_RAM_read_flow = 20; // segmentation fault checks prevent access
      for (p = 0; p < size_of_physical_memory_in_words; p = p + 1) {
        nid_pc_lw_at_p eq 1 nid_pc_lw_add nid_p_virtual_address
        nid_pc_lw_from_p ite 2 nid_pc_lw_at_p nid_p_RAM_word nid_RAM_read_flow
        nid_RAM_read_flow = nid_pc_lw_from_p;
      }
      nid_pc_lw_val ite 2 nid_pc_flag nid_RAM_read_flow nid_register_flow[rd]
      nid_register_flow[rd] = nid_pc_lw_val;
    }
  } else if (pc: sw rs1,rs2,imm) {
    nid_pc_sw_imm constd 2 imm
    nid_pc_sw_add add 2 nid_rs1_register nid_pc_sw_imm
    nid_pc_sw_acc ite 2 nid_pc_flag nid_pc_sw_add nid_start_access_flow
    nid_start_access_flow = nid_pc_sw_acc;
    for (p = 0; p < size_of_physical_memory_in_words; p = p + 1) {
      nid_pc_sw_at_p eq 1 nid_pc_sw_add nid_p_virtual_address
      nid_pc_sw_to_p ite 2 nid_pc_sw_at_p nid_rs2_register nid_RAM_write_flow[p]
      nid_pc_sw_into_p ite 2 nid_pc_flag nid_pc_sw_to_p nid_RAM_write_flow[p]
      nid_RAM_write_flow[p] = nid_pc_sw_into_p;
    }
  } else if (pc: beq rs1,rs2,imm) {
    nid_pc_beq eq 1 nid_rs1_register nid_rs2_register
    nid_pc_neq not 1 nid_pc_beq
  } else if (pc: jal rd,imm) {
    if (rd != zero) {
      nid_pc_link constd 2 pc+4
      nid_pc_ret ite 2 nid_pc_flag nid_pc_link nid_register_flow[rd]
      nid_register_flow[rd] = nid_pc_ret;
    }
  } else if (pc: jalr rd,rs1,imm) {
    // rd update unsupported
  } else if (pc: ecall) {
    nid_pc_df0 ite 1 nid_pc_flag 11 nid_ecall_flow
    nid_ecall_flow = nid_pc_df0;
  }
}
```

Fig. 17. 32-bit BEATOR data flow part II

```
current_callee = start_of_code_segment;
for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4) {
  if (pc: [lui|add|sub|mul|divu|remu|sltu|lw|sw])
    control_in[pc+4] =
      new control_out([lui|add|sub|mul|divu|remu|sltu|lw|sw], pc, 0, control_in[pc+4]);
  else if (pc: addi rd,rs1,imm) {
    if (rs1 == zero)
      if (imm != 0)
        if (rd == a7)
          // assert: next instruction is ecall
          reg_a7 = imm;
    control_in[pc+4] = new control_out(addi, pc, 0, control_in[pc+4]);
  } else if (pc: beq rs1,rs2,imm) {
    control_in[pc+imm] = new control_out(beq, pc, nid_pc_beq, control_in[pc+imm]);
    control_in[pc+4] = new control_out(beq, pc, nid_pc_neq, control_in[pc+4]);
  } else if (pc: jal rd,imm) {
    control_in[pc+imm] = new control_out(jal, pc, 0, control_in[pc+imm]);
    if (rd != zero)
      control_in[pc+4] =
        new control_out(jalr, pc+imm, nid_pc_link, control_in[pc+4]);
  } else if (pc: jalr rd,rs1,imm) {
    if (rd == zero)
      if (imm == 0)
        if (rs1 == ra) {
          // assert: current callee points to an instruction to which a jal jumps
          call_return[current_callee] = pc;
          // assert: next "procedure body" begins right after jalr
          current_callee = pc+4;
        }
  } else if (pc: ecall) {
    if (reg_a7 == SYSCALL_EXIT)
      // assert: exit ecall is immediately followed by first "procedure body" in code
      current_callee = pc+4;
    reg_a7 = 0;
    control_in[pc+4] = new control_out(ecall, pc, 0, control_in[pc+4]);
  }
}
```

Fig. 18. 32-bit BEATOR control flow preparation

```
; syscalls
nid_syscall_id_exit    constd 2 93 ; SYSCALL_EXIT
nid_syscall_id_read    constd 2 63 ; SYSCALL_READ
nid_syscall_id_write   constd 2 64 ; SYSCALL_WRITE
nid_syscall_id_openat  constd 2 56 ; SYSCALL_OPENAT
nid_syscall_id_brk     constd 2 214 ; SYSCALL_BRK

nid_exit_syscall    eq 1 nid_a7_register nid_syscall_id_exit ; a7 == SYSCALL_EXIT
nid_read_syscall    eq 1 nid_a7_register nid_syscall_id_read ; a7 == SYSCALL_READ
nid_write_syscall   eq 1 nid_a7_register nid_syscall_id_write ; a7 == SYSCALL_WRITE
nid_openat_syscall  eq 1 nid_a7_register nid_syscall_id_openat ; a7 == SYSCALL_OPENAT
nid_brk_syscall     eq 1 nid_a7_register nid_syscall_id_brk ; a7 == SYSCALL_BRK

nid_exit_active and 1 nid_ecall_flow nid_exit_syscall ; exit ecall is active
; stay in kernel mode indefinitely if exit ecall is active
nid_exit_kernel ite 1 60 nid_exit_syscall nid_exit_active
nid_kernel_mode_flow = nid_exit_kernel;
```

Fig. 19. 32-bit BEATOR system call header and `exit` system call

```
nid_read_active and 1 nid_ecall_flow nid_read_syscall ; read ecall is active
; a1 is start address of buffer for checking address validity
nid_read_access ite 2 nid_read_active nid_a1_register nid_start_access_flow
nid_start_access_flow = nid_read_access;

; go into kernel mode if read ecall is active
nid_read_kernel ite 1 nid_read_active 11 nid_kernel_mode_flow
nid_kernel_mode_flow = nid_read_kernel;

; set a0 = 0 bytes if read ecall is active
nid_read_set_a0 ite 2 nid_read_active 20 nid_register_flow{a0}
nid_register_flow{a0} = nid_read_set_a0;

nid_read_sub    sub 2 nid_a2_register nid_a0_register ; a2 - a0
nid_read_cmp    ugte 1 nid_read_sub 24 ; a2 - a0 >= 4 bytes
; read 4 bytes if a2 - a0 >= 4 bytes, or else a2 - a0 bytes
nid_read_inc    ite 2 nid_read_cmp 24 nid_read_sub
nid_read_inc2   eq 1 nid_read_inc 22 ; increment == 2
; unsigned-extended 2-byte input if increment == 2, or else unsigned-extended 1-byte input
nid_read_input2 ite 2 nid_read_inc2 92 91
nid_read_inc3   eq 1 nid_read_inc 23 ; increment == 3
; unsigned-extended 3-byte input if increment == 3
nid_read_input3 ite 2 nid_read_inc3 93 nid_read_input2
nid_read_inc4   eq 1 nid_read_inc 24 ; increment == 4
nid_read_input4 ite 2 nid_read_inc4 94 nid_read_input3 ; 4-byte input if increment == 4
nid_read_cursor add 2 nid_a1_register nid_a0_register ; a1 + a0
nid_read_more   ult 1 nid_a0_register nid_a2_register ; a0 < a2
nid_read_goon   and 1 nid_read_syscall nid_read_more ; a7 == SYSCALL_READ and a0 < a2

nid_read_active_kernel and 1 60 nid_read_goon ; read ecall is in kernel mode and not done yet
nid_read_ugti          ugt 1 nid_read_inc 20 ; increment > 0
; read ecall is in kernel mode and not done yet and increment > 0
nid_read_kernel_inc    and 1 nid_read_active_kernel nid_read_ugti

for (p = 0; p < size_of_physical_memory_in_words; p = p + 1) {
  nid_read_at_p    eq 1 nid_read_cursor nid_p_virtual_address
  nid_read_input_p ite 2 nid_read_at_p nid_read_input4 nid_RAM_write_flow[p]
  ; read input into RAM[a1 + a0]
  nid_read_into_p  ite 2 nid_read_kernel_inc nid_read_input_p nid_RAM_write_flow[p]
  nid_RAM_write_flow[p] = nid_read_into_p;
}

nid_read_move_cursor add 2 nid_a0_register nid_read_inc ; a0 + increment
; set a0 = a0 + increment if read ecall is in kernel mode and not done yet
nid_read_set_cursor  ite 2 nid_read_active_kernel nid_read_move_cursor nid_register_flow{a0}
nid_register_flow{a0} = nid_read_set_cursor;

; stay in kernel mode if read ecall is in kernel mode and not done yet
nid_read_stay ite 1 nid_read_active_kernel 11 nid_kernel_mode_flow
nid_kernel_mode_flow = nid_read_stay;
```

Fig. 20. 32-bit BEATOR read system call

```
nid_write_active and 1 nid_ecall_flow nid_write_syscall ; write ecall is active
; a1 is start address of buffer for checking address validity
nid_write_access ite 2 nid_write_active 211 nid_start_access_flow
nid_start_access_flow = nid_write_access;

; set a0 = a2 if write ecall is active
nid_write_set_a0 ite 2 nid_write_active nid_a2_register nid_register_flow{a0}
nid_register_flow{a0} = nid_write_set_a0;


nid_openat_active and 1 nid_ecall_flow nid_openat_syscall ; openat ecall is active
; a1 is start address of buffer for checking address validity
nid_openat_access ite 2 nid_openat_active nid_a1_register nid_start_access_flow
nid_start_access_flow = nid_openat_access;

nid_openat_bump   state 2 fd-bump-pointer
; initial fd-bump-pointer is 1 (file descriptor bump pointer)
nid_openat_init   init 2 nid_openat_bump 21
nid_openat_inc    inc 2 nid_openat_bump
; fd-bump-pointer + 1 if openat ecall is active
nid_openat_newfd  ite 2 nid_openat_active nid_fd_inc nid_fd_bump
; increment fd-bump-pointer if openat ecall is active
nid_openat_next   next 2 nid_fd_bump nid_openat_newfd
; set a0 = fd-bump-pointer + 1 if openat ecall is active
nid_openat_set_a0 ite 2 nid_openat_active nid_fd_inc nid_register_flow{a0}
nid_register_flow{a0} = nid_openat_set_a0;


nid_brk_active  and 1 nid_ecall_flow nid_brk_syscall ; brk ecall is active
nid_brk_bump    state 2 brk-bump-pointer
nid_brk_init    init 2 nid_brk_bump 33 ; current program break
nid_brk_shrink  ulte 1 nid_brk_bump nid_a0_register ; brk <= a0
nid_brk_free    ult 1 nid_a0_register nid_sp_register ; a0 < sp
nid_brk_segment and 1 nid_brk_shrink nid_brk_free ; brk <= a0 < sp
nid_brk_reset   and 2 nid_a0_register 23 ; reset all but 2 LSBs of a0
nid_brk_aligned eq 1 nid_brk_reset 20 ; 2 LSBs of a0 == 0 (a0 is word-aligned)
; brk <= a0 < sp and a0 is word-aligned (a0 is valid)
nid_brk_valid   and 1 nid_brk_segment nid_brk_aligned
nid_brk_ok      and 1 nid_brk_active nid_brk_valid ; brk ecall is active and a0 is valid
; brk = a0 if brk ecall is active and a0 is valid
nid_brk_newbrk  ite 2 nid_brk_ok nid_a0_register nid_brk_bump
; set brk = a0 if brk ecall is active and a0 is valid
nid_brk_next    next 2 nid_brk_bump nid_brk_newbrk
nid_brk_invalid not 1 nid_brk_valid ; a0 is invalid
nid_brk_fail    and 1 nid_brk_active nid_brk_invalid ; brk ecall is active and a0 is invalid
; set a0 = brk if brk ecall is active and a0 is invalid
nid_brk_set_a0 ite 2 nid_brk_fail nid_brk_bump nid_register_flow{a0}
nid_register_flow{a0} = nid_brk_set_a0;

nid_update_kernel next 1 60 nid_kernel_mode_flow ; updating kernel-mode flag
```

Fig. 21. 32-bit BEATOR write, openat, and brk system calls

```
; control flow
for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4) {
  nid_control_flow = 10;
  control_out = control_in[pc];
  while (control_out != 0) {
    (from_is, from_pc, nid_condition, control_out) = control_out;
    if (from_pc != 0) {
      if (from_is == beq) {
        nid_pc_active and 1 nid_from_pc_flag nid_condition
        nid_from_active = nid_pc_active;
      } else if (from_is == jalr) {
        from_pc = call_return[from_pc];
        nid_pc_not   not 2 21
        nid_pc_lsbrst and 2 nid_ra_register nid_pc_not
        nid_pc_equal eq 1 nid_pc_lsbrst nid_condition
        nid_pc_active and 1 nid_from_pc_flag nid_pc_equal
        nid_from_active = nid_pc_active;
      } else if (from_is == ecall) {
        nid_pc_kernel  state 1 kernel-mode-pc-flag-from_pc
        nid_pc_inactive init 1 nid_pc_kernel 10 ; ecall is initially inactive
        ; activate ecall and keep active while in kernel mode
        nid_pc_active  ite 1 nid_pc_kernel 60 nid_from_pc_flag
        ; keep ecall active while in kernel mode
        nid_pc_stay    next 1 nid_pc_kernel nid_pc_active
        ; ecall is active but not in kernel mode anymore
        nid_pc_leave   and 1 nid_pc_kernel 62
        nid_from_active = nid_pc_leave;
      } else
        nid_from_active = nid_from_pc_flag;
      if (nid_control_flow == 10)
        nid_control_flow = nid_from_active;
      else {
        nid_pc_next ite 1 nid_from_active 11 nid_control_flow
        nid_control_flow = nid_pc_next;
      }
    }
  }
  nid_pc_next next 1 nid_pc_flag nid_control_flow
}
```

Fig. 22. 32-bit BEATOR control flow

```
; updating registers
for (r = 1; r < 32; r++)
  nid_r_register_update next 2 nid_r_register nid_register_flow[r] register_name[r]

; updating physical memory
for (p = 0; p < size_of_physical_memory_in_words; p = p + 1)
  nid_p_memory_update next 2 nid_p_RAM_word nid_RAM_write_flow[p] RAM-word-p
```

Fig. 23. 32-bit BEATOR register and memory update

```
; checking syscall id
nid_check_syscall0 not 1 nid_syscall_id_exit ; a7 != SYSCALL_EXIT
nid_check_syscall1 not 1 nid_syscall_id_read ; a7 != SYSCALL_READ
nid_check_syscall2 not 1 nid_syscall_id_write ; a7 != SYSCALL_WRITE
nid_check_syscall3 not 1 nid_syscall_id_openat ; a7 != SYSCALL_OPENAT
nid_check_syscall4 not 1 nid_syscall_id_brk ; a7 != SYSCALL_BRK
nid_check_syscall5 and 1 nid_check_syscall0 nid_check_syscall1 ; & a7 != SYSCALL_READ
nid_check_syscall6 and 1 nid_check_syscall5 nid_check_syscall2 ; & a7 != SYSCALL_WRITE
nid_check_syscall7 and 1 nid_check_syscall6 nid_check_syscall3 ; & a7 != SYSCALL_OPENAT
nid_check_syscall8 and 1 nid_check_syscall7 nid_check_syscall4 ; & a7 != SYSCALL_BRK
; ecall is active for invalid syscall id
nid_check_syscall9 and 1 nid_ecall_flow nid_check_syscall8
nid_check_syscall10 bad nid_check_syscall9 b0 ; invalid syscall id

; checking exit code
nid_check_exit_code0 neq 1 nid-a0-register 20 ; a0 != zero exit code
; exit ecall is active with non-zero exit code
nid_check_exit_code1 and 1 nid_exit_active nid_check_exit_code0
nid_check_exit_code2 bad nid_check_exit_code1 b1 ; non-zero exit code

; checking division and remainder by zero
nid_check_division0  eq 1 nid_division_flow 20
nid_check_division1  bad nid_check_division0 b2 ; division by zero
nid_check_remainder0 eq 1 nid_remainder_flow 20
nid_check_remainder1 bad nid_check_remainder0 b3 ; remainder by zero

; checking address validity
; is start address of memory access word-aligned?
nid_check_address0 and 2 nid_start_access_flow 23 ; reset all but 2 LSBs of address
; 2 LSBs of address != 0 (address is not word-aligned)
nid_check_address1 neq 1 nid_check_address0 20
nid_check_address2 bad nid_check_address1 b4 ; word-unaligned memory access
```

Fig. 24. 32-bit BEATOR bad states other than segmentation faults

```
; checking segmentation faults
; is start address of memory access in a valid segment?
nid_check_segfault0  ult 1 nid_start_access_flow 30 ; address < start of data segment
nid_check_segfault1  bad nid_check_segfault0 b6 ; memory access below data segment
nid_check_segfault2  ugte 1 nid_start_access_flow 31 ; address >= end of data segment
nid_check_segfault3  ult 1 nid_start_access_flow 32 ; address < start of heap segment
nid_check_segfault4  and 1 nid_check_segfault2 nid_check_segfault3
; memory access in between data and heap segments
nid_check_segfault5  bad nid_check_segfault4 b7
; address >= current end of heap segment
nid_check_segfault6  ugte 1 nid_start_access_flow nid_bump_pointer
; address < current start of stack segment
nid_check_segfault7  ult 1 nid_start_access_flow nid-sp-register
nid_check_segfault8  and 1 nid_check_segfault6 nid_check_segfault7
; memory access in between current heap and stack segments
nid_check_segfault9  bad nid_check_segfault8 b8
; address >= allowed end of heap segment
nid_check_segfault10 ugte 1 nid_start_access_flow 34
; address < current end of heap segment
nid_check_segfault11 ult 1 nid_start_access_flow nid_bump_pointer
nid_check_segfault12 and 1 nid_check_segfault10 nid_check_segfault11
; memory access in between allowed and current end of heap segment
nid_check_segfault13 bad nid_check_segfault12 b9
; address >= current start of stack segment
nid_check_segfault14 ugte 1 nid_start_access_flow nid-sp-register
; address < allowed start of stack segment
nid_check_segfault15 ult 1 nid_start_access_flow 35
nid_check_segfault16 and 1 nid_check_segfault14 nid_check_segfault15
; memory access in between current and allowed start of stack segment
nid_check_segfault17 bad nid_check_segfault16 b10
; address > highest address in 32-bit virtual address space (4GB)
nid_check_segfault18 ugt 1 nid_start_access_flow 50
; memory access above stack segment
nid_check_segfault19 bad nid_check_segfault18 b11
```

Fig. 25. 32-bit BEATOR segmentation fault checks

```
initialize QUBO model Q with O(m+|P|) init lines in BTOR2 model B

// for all n state transitions:
for (i = 1; i <= n; i++) {

  // updating openat fd bump pointer
  nid_openat_next next 2 nid_fd_bump nid_openat_newfd in BTOR2 model B:
    bit-blast nid_openat_newfd into O(w) BQ function

  // updating brk bump pointer
  nid_brk_next next 2 nid_brk_bump nid_brk_newbrk in BTOR2 model B:
    bit-blast nid_brk_newbrk into O(w) BQ function

  // updating kernel-mode flag
  nid_update_kernel next 1 60 nid_kernel_mode_flow in BTOR2 model B:
    bit-blast nid_kernel_mode_flow into O(w) BQ function

  // updating control flow
  // the total size of the pc-flag BQ functions is still only: O(w·|P|)
  for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4)
    nid_pc_next next 1 nid_pc_flag nid_control_flow in BTOR2 model B:
      bit-blast nid_control_flow into O(w·|P|) BQ function
    nid_pc_stay next 1 nid_pc_kernel nid_pc_active in BTOR2 model B:
      bit-blast nid_pc_active into O(w·|P|) BQ function

  // updating registers
  for (r = 1; r < 32; r++)
    nid_r_register_update next 2 nid_r_register nid_register_flow[r] in BTOR2 model B:
      bit-blast nid_register_flow[r] into O(m·w²·|P|) BQ function

  // updating physical memory
  for (p = 0; p < m; p = p + 1)
    nid_p_memory_update next 2 nid_p_RAM_word nid_RAM_write_flow[p] in BTOR2 model B:
      bit-blast nid_RAM_write_flow[p] into O(w²·|P|) BQ function

  // checking error states
  for (b = 0; b < 12; b++)
    nid_check_* bad nid_condition bb in BTOR2 model B:
      bit-blast nid_condition into O(w·|P|) BQ function

  compose O(m·w²·|P|) BQ function
}

compose O(n·m·w²·|P|) QUBO model Q
```

Fig. 26. QUBOT algorithm

```
initialize quantum circuit C with O(m+|P|) init lines in BTOR2 model B

// for all n state transitions:
for (i = 1; i <= n; i++) {

  // updating openat fd bump pointer
  nid_openat_next next 2 nid_fd_bump nid_openat_newfd in BTOR2 model B:
    bit-blast nid_openat_newfd into O(w) quantum circuit

  // updating brk bump pointer
  nid_brk_next next 2 nid_brk_bump nid_brk_newbrk in BTOR2 model B:
    bit-blast nid_brk_newbrk into O(w) quantum circuit

  // updating kernel-mode flag
  nid_update_kernel next 1 60 nid_kernel_mode_flow in BTOR2 model B:
    bit-blast nid_kernel_mode_flow into O(w) quantum circuit

  // updating control flow
  // the total size of the pc-flag BQ functions is still only: O(w·|P|)
  for (pc = start_of_code_segment; pc < end_of_code_segment; pc = pc + 4)
    nid_pc_next next 1 nid_pc_flag nid_control_flow in BTOR2 model B:
      bit-blast nid_control_flow into O(w·|P|) quantum circuit
    nid_pc_stay next 1 nid_pc_kernel nid_pc_active in BTOR2 model B:
      bit-blast nid_pc_active into O(w·|P|) quantum circuit

  // updating registers
  for (r = 1; r < 32; r++)
    nid_r_register_update next 2 nid_r_register nid_register_flow[r] in BTOR2 model B:
      bit-blast nid_register_flow[r] into O(m·w·|P|) quantum circuit

  // updating physical memory
  for (p = 0; p < m; p = p + 1)
    nid_p_memory_update next 2 nid_p_RAM_word nid_RAM_write_flow[p] in BTOR2 model B:
      bit-blast nid_RAM_write_flow[p] into O(w·|P|) quantum circuit

  // checking error states
  for (b = 0; b < 12; b++)
    nid_check_* bad nid_condition bb in BTOR2 model B:
      bit-blast nid_condition into O(w·|P|) quantum circuit

  compose O(m·w·|P|) quantum circuit
}

compose O(n·m·w·|P|) quantum circuit C
```

Fig. 27. QUARC algorithm