SMT - Bit Vectors

Florian Märkl

June 26, 2020

Abstract

Due to how real computers are designed, SMT formulas built on unbounded integers are generally unsuited for deciding properties on program code and instead, sequences of bits, called *bit vectors*, are used to represent machine integers. After establishing a calculus for SMT formulas on such data, this document elaborates on how these can be automatically decided by using the *flattening* approach to translate them into propositional logic, which can be solved by any SAT solver. This approach is extended to *incremental flattening*, which is able to decide certain types of formulas significantly faster. For these processes, an exemplarily implementation in Haskell is developed, which is finally being used to solve a small CrackMe-challenge as an example application.

1 Motivation

Satisfiability modulo theories (SMT), the problem of deciding the satisfiability of a formula in first-order logic, plays a major role in the field of program analysis where the semantics of real machine code is transformed in a way such that an SMT solver can decide certain properties of the underlying code. Coming from a mathematical background, SMT formulas are commonly built upon data types such as unbounded natural numbers, however this is often not suitable for real-world machine code, as the following example demonstrates:

```
uint8_t a = 200;
uint8_t b = a + 98;
assert(b > a);
```

If `a` and `b` were able to hold any natural number, the assertion `b > a` would obviously hold, but since this code only uses 8-bit unsigned integers, an overflow occurs in the addition, `b` gets assigned the value `42` and the assertion fails.

In order to be able to accurately represent such semantics in SMT, formulas based on sequences of bits with a fixed length, called *bit vectors*, can be used. We define one variant of syntax and semantics for such formulas, then depict how satisfiability for them can be decided using the *flattening* approach while developing an example implementation in Haskell. Finally, we practically use our implementation to solve a small "Capture The Flag"-like CrackMe challenge.

1.1 Info about Code Listings

The code shown in this document is heavily simplified Haskell-like pseudocode, but closely follows the structure of the accompanying implementation. Namings of functions and variables have been kept identical wherever appropriate to allow conveniently finding the correspondents in the real code.

2 Definitions

Our syntax for bit vector formulas closely follows the definitions given in [4], however we will slightly deviate from their semantics.

$$\begin{array}{rcl}
formula & \rightarrow & formula \wedge formula \mid \neg formula \mid (formula) \mid atom \\
atom & \rightarrow & term < term \mid term = term \mid term[constant] \\
term & \rightarrow & term\ op\ term \mid var{-}identifier \mid {\sim}term \mid constant \\
 & & \mid atom\ ?\ term : term \mid term[constant : constant] \mid \mathrm{ext}\,(term) \\
op & \rightarrow & + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \,|\, \mid \oplus \mid \circ
\end{array}$$

It is crucial to understand the idea behind the separation into *term*, *atom* and *formula* as it is essential to the solving approach later:

Term is any construct that would yield a bitvector if it were evaluated.

Atom is a construct that yields a boolean value, but by itself is not built by combination of other boolean operations and is thus indivisible ("atomic") with respect to propositional logic.

Formula is the top-level entity that describes one entire problem to solve.

As already hinted by this distinction, the calculus is statically typed. While formulas and atoms are boolean, every term has a bit vector type consisting of a fixed size (i.e. the count of bits in the vector) and the information of whether the value is interpreted as signed or unsigned. Where not directly obvious, we explicitly specify the type of an expression a by e.g. a_{u32} or a_{s64} where u and s stand for signed and unsigned, and the number specifies the size of the vector. To directly use the size of an expression a as a number, we write size(a).

2.1 Semantics

When interpreted as integer values, bit vectors express their value in Base-2. Signed vectors use the two's complement as this is the de-facto standard of encoding numbers in binary.

Semantics of operations on such values are defined as follows. Binary operators are only defined when both operands have the same type, unless explicitly stated otherwise.

Syntax	Semantics	Resulting Type
$a \wedge b$ $\neg a$ $a < b$ $a = b$	Common meaning of conjunction, negation, comparison and equality. $<$ also considers the signedness of its operands.	Boolean.
$a[i]$	Picks the i-th bit out of a and interprets it as boolean.	Boolean.
$a \mid b$ $a \,\&\, b$ $a \oplus b$	Bitwise operators for or, and and xor.	Same bit vector type as operands.
${\sim}a$	Complement, i.e. negation of all bits in a.	Same bit vector type as a.
$a \ll b$ $a \gg b$	Shift the bits in a by b digits to the left or right, respectively, filling up with zeroes. To enforce a meaningful definition of the result, we restrict the sizes of a and b to size(a) $= 2^{\mathrm{size}(b)}$, as well as only allow unsigned types for b. This means that b's type covers exactly the range of numbers addressing bits in a.	Same bit vector type as a.

$a + b$ $a - b$ $a \cdot b$ $a \,/\, b$	Arithmetic operations of addition, subtraction, multiplication and division, respecting the signedness of their operands.	Same bit vector type as operands.
$\text{ext}(a)$	Extends a to a larger size, preserving its interpreted integer value. Newly added higher bits are filled with zeroes if a is unsigned or repeatedly $a[\text{size}(a)-1]$ if it is signed.	Bit vector of the same signedness as a. The size must be explicitly specified.
$a \circ b$	Concatenation of a and b where a fills the lower and b the higher bits.	Bit vector of size $\text{size}(a)+\text{size}(b)$. The signedness must be explicitly specified.
$a[b\!:c]$	Slice, i.e. a sub-vector of a where b specifies the offset and and c the size in a.	Bit vector of size c. The signedness must be explicitly specified.
$c?\,a\!:b$	Ternary operator, takes the value of a if the atom c evaluates to true, or b otherwise.	Same bit vector type as a and b.

3 Flattening

One approach to decide bit vector formulas is called *bit-blasting* or *flattening* and is specified in [4]. The idea for deciding a bit vector formula f is to generate a formula p in propositional logic, i.e. one consisting only of boolean entities, that is *equisatisfiable* to f. This means that p is satisfiable if and only if f is satisfiable. This propositional formula can then be decided by any generic SAT solver. Furthermore, p is generated in a way that directly maps some of its propositional variables to bits of the original bit vector variables in f, so when a concrete satisfying assignment for p has been found, a satisfying assignment for f can be directly inferred.

3.1 High-Level Algorithm

The algorithm starts by collecting all atoms and terms occuring in a given formula f, including any term being part of another, as well as atoms being part of any ternary operator term. It then reserves a single propositional variable for each atom and every bit of each term.

The propositional formula p is then constructed upon these variables as a conjunction of the following constraints derived from f:

- One for each term appearing anywhere in f, restricting its bits with respect to its definition and children.
- One for each atom appearing anywhere in f, restricting its value with respect to its definition and children.
- One, called *skeleton*, for the overall formula, bringing its atoms in relation.

Figure 1 illustrates the resulting relations between terms, atoms, constraints and propositional variables for an example formula.

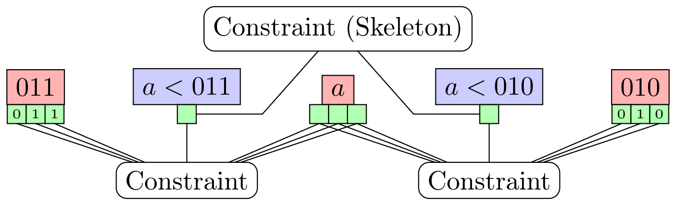

Figure 1: Abstract illustration of the reserved propositional variables (green) per atom (blue) and term (red), along with the constraints imposed by the flattening algorithm when deciding the formula $a < 011 \land \neg a < 010$.

This process can be expressed formally as the following pseudocode, which our Haskell implementation is directly based on:

```
flatten :: Formula -> Propositional.Formula
flatten f =
    let
        termProps       = reserveVarsForAll (terms f)
        atomProps       = reserveVarsForAll (atoms f)
        termConstraints = {termConstraint atomProps termProps term | term ∈ (terms f)}
        atomConstraints = {atomConstraint atomProps termProps atom | atom ∈ (atoms f)}
        skel            = skeleton atomProps f
    in (skel ∪ termConstraints ∪ atomConstraints)
```

Finally, we can solve the generated formula with a SAT solver and derive values for our bit vector variables. We use the minisat-solver [6] package, which exposes bindings to minisat [5].

```
solve :: Formula -> SolveResult
solve f =
    let
        flat = flatten f
        satSolution = SAT.MiniSat.solve flat
    in reconstructResult satSolution
```

What is left now is to construct the skeleton and constraints for atoms and terms in a way that they correctly represent the semantics specified in section 2.1.

3.2 Skeleton

The skeleton is constructed trivially from the top-level formula structure by substituting atoms by their reserved single propositional variable and converting negation and conjunction to their correspondents in propositional logic, which yields three cases in the following pseudocode:

```
skeleton :: (Map Atom -> Propositional.Variable) -> Formula -> Propositional.Formula
skeleton atomProps (Atom atom) = atomProps[atom]
skeleton atomProps (¬f)        = ¬(skeleton atomProps f)
skeleton atomProps (l ∧ r)     = (skeleton atomProps l) ∧ (skeleton atomProps r)
```

3.3 Term and Atom Constraints

Constraints for terms and atoms are more involved than the skeleton. In essence, they directly describe the semantics of the respective bit vector operations on the bits of their operands in the propositional logic theory.

3.3.1 Bitwise Operations and Equality

Bitwise operations, i.e. and, or, xor and complement, are fairly straightforward to express as the value of each bit of the resulting term is dependent only on one known bit of all operands. As an example, we can implement bitwise xor by the following constraint:

$$(l \oplus r)[i] \iff (l[i] \oplus r[i]) \tag{1}$$

Equality follows a similar scheme as bitwise operations, but combines all compared bits to a single conjunction:

$$(l = r) \iff \bigwedge_i (l[i] \iff r[i]) \tag{2}$$

3.3.2 Addition, Subtraction and Comparison

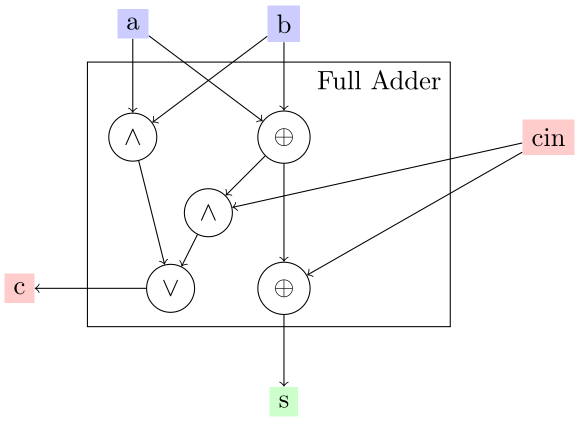

Figure 2: Full adder circuit, adding two bits together while considering an incoming carry and producing a result and outgoing carry bit.

Addition can be implemented by combining basic logic circuits, as also commonly found in electronics. We start by defining a *full adder*, which takes two input bits a and b along with one incoming carry bit cin and outputs the resulting sum s and carry bit c, depicted in fig. 2 and written as logic functions as:

$$\begin{aligned} \mathrm{s}(a, b, cin) &= cin \oplus (a \oplus b) \\ \mathrm{c}(a, b, cin) &= (a \wedge b) \vee ((a \oplus b) \wedge cin) \end{aligned} \tag{3}$$

Multiple full adder circuits can eventually be combined into a *ripple-carry adder* for operating on more than one digit, as seen in fig. 3. Directly translating this circuit into formulas in propositional logic yields:

$$\begin{aligned} \mathrm{carry}(l, r, cin)[0] &= cin \\ \mathrm{carry}(l, r, cin)[i+1] &= cout(l[i+1], r[i+1], \mathrm{carry}(cin, l, r)[i]) \\ \mathrm{sum}(l, r, cin)[i] &= \mathrm{s}(l[i], r[i], \mathrm{carry}(l, r, cin)[i]) \end{aligned} \tag{4}$$

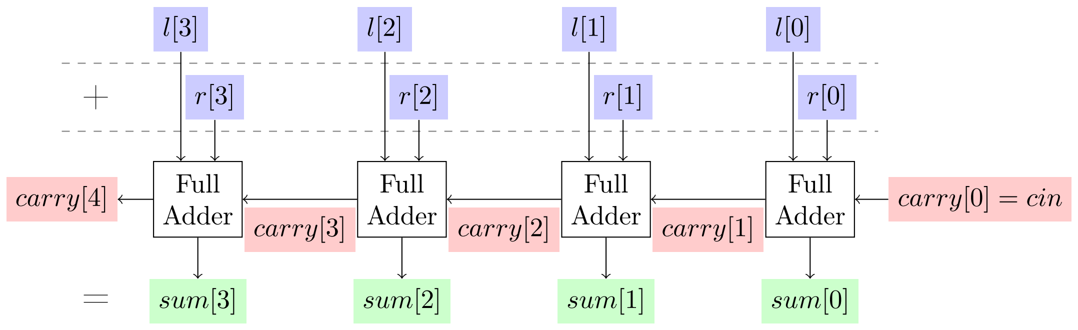

Figure 3: Ripple-carry adder, combining multiple full adder circuits to add bit vectors of arbitrary size.

This definition now allows us to derive a constraint for addition:

$$(l + r)[i] \iff \mathrm{sum}(l, r, 0)[i] \tag{5}$$

Defining the constraint for each output bit like this means bit i recurses with depth i in $\mathrm{carry}(l, r, cin)$, yielding a formula of size $O(n)$ per bit and thus size $O(n^2)$ for all bits combined when n is the total number of bits. By introducing propositional variables for a helper bit vector k where

$$\begin{gathered} k[0] \iff cin \\ k[i+1] \iff \mathrm{c}(l[i+1], r[i+1], k[i]) \end{gathered} \tag{6}$$

we can avoid repeatedly writing down the unfolded recursion for the carry of bit i, but can instead directly refer to $k[i]$. This way, the complexity for n bit constraints can be reduced to $O(n)$ by using the following alternative definition, since it allows us to establish a constant uppper bound for the size of formulas needed for each bit, regardless of the total number of bits.

$$(l + r)[i] \iff \mathrm{s}(l[i], r[i], k[i]) \tag{7}$$

For subtraction, we exploit the fact that the negation of an integer number stored in a bit vector is always its two's complement, i.e. $-r = {\sim}r + 1$ holds. This means we can perform subtraction as $l - r = l + {\sim}r + 1$. Furthermore, we take advantage of the fact that the *cin* bit of an addition allows us to increase the result by one, so we can perform the subtraction using only a single addition pass:

$$(l - r)[i] \iff \mathrm{sum}(l, {\sim}r, 1)[i] \tag{8}$$

Both addition and subtraction are identical for signed and unsigned operands.

The less-than operator can be based on subtraction as we know that $l < r \iff (l - r) < 0$. Here we make use of the additional, final *cout* bit coming out of the addition performed for the subtraction. In essence, it indicates that an underflow has occured during the subtraction, which directly corresponds to $(l - r) < 0$. Unsigned comparison is hence implemented as

$$\begin{gathered} l_u < r_u \iff \mathrm{carry}(l, {\sim}r, 1)[sz] \\ \text{where } sz = \mathrm{size}(l) = \mathrm{size}(r) \end{gathered} \tag{9}$$

while signed comparison, where extra care must be taken for the signedness of its operands, can be implemented as

$$\begin{aligned}&l_s < r_s \iff \text{carry}(l, {\sim}r, 1)[sz] \oplus (l[sz-1] \iff r[sz-1]) \\ &\text{where } sz = \text{size}(l) = \text{size}(r)\end{aligned} \tag{10}$$

3.3.3 Shifts

We start by defining *static* shifts $\ll_{static}$ and $\gg_{static}$, i.e. ones where the shift distance C is known at flattening-time:

$$\begin{aligned}(l \ll_{static} C)[i] &= \begin{cases} l[i-C] & \text{if } i - C \geq 0 \\ 0 & \text{otherwise}\end{cases} \\ (l \gg_{static} C)[i] &= \begin{cases} l[i+C] & \text{if } i + C < \text{size}(l) \\ 0 & \text{otherwise}\end{cases}\end{aligned} \tag{11}$$

Because the operands of $l \ll r$ have been limited by $\text{size}(l) = 2^{\text{size}(r)}$ in section 2.1, we can now construct a so-called *barrel shifter* for arbitrary shifts. Let $w = \text{size}(r)$ be the size of the shift distance, then we split the entire shift operation into w stages, where stage s can either shift the value by 2^s bits, or leave it unaltered. The criteria for whether to shift in stage s is directly given by $r[s]$. More formally, for a left shift:

$$\begin{aligned}\text{lshift}(l, r, -1)[i] &= l[i] \\ \text{lshift}(l, r, s)[i] &= \begin{cases} (\text{lshift}(l, r, s-1) \ll_{static} s)[i] & \text{if } r[s] \\ \text{lshift}(l, r, s-1)[i] & \text{if } \neg r[s]\end{cases} \\ (l \ll r)[i] &\iff \text{lshift}(l, r, \text{size}(r) - 1)[i]\end{aligned} \tag{12}$$

Right shift is defined analogously by replacing $\ll_{static}$ by $\gg_{static}$.

3.3.4 Multiplication and Division

Because multiplication is distributive over addition, we are allowed to multiply the left operand by each bit of the right operand individually and then add up the result to obtain the full multiplication result. Since multiplication by a bit vector with only a single bit set to one has the same effect as a left shift by the index of this bit, we can simply make use of the already established static shift for it. This approach leads to a similar recursive definition as for dynamic shifts:

$$\begin{aligned}&\text{mul}(l, r, -1) = 0 \\ &\text{mul}(l, r, s) = mul(l, r, s-1) + \begin{cases} (l \ll_{static} s) & \text{if } r[s] \\ 0 & \text{if } \neg r[s]\end{cases} \\ &(l \cdot r)[i] \iff \text{mul}(l, r, \text{size}(r) - 1)[i]\end{aligned} \tag{13}$$

Multiplication is defined in the same way for both signed and unsigned operands.

All previous constraints were expressed as direct material equivalences for each resulting bit. For division, we instead use the result of the division as an inner part of a multiplication and addition that would produce the given dividend l. We also reserve additional propositional variables to represent the remainder rem of the division and constraint it in a way that the division result is unique and correct. For unsigned division, these two constraints are:

$$l = (l/r) \cdot r + rem \tag{14}$$

$$rem < r \tag{15}$$

However, implementing this constraint directly on the original types of l and r is not restrictive enough. Consider the following example:

$$\begin{aligned} &4_{u8} = 172_{u8} \cdot 3_{u8} + 0_{u8} \\ &0_{u8} < 3_{u8} \end{aligned} \tag{16}$$

Because of an overflow occuring in the multiplication, these statements are true, which means for the division $4_{u8}/3_{u8}$, we would accept the result 172_{u8} which obviously does not fit our understanding of division. Thus, to avoid such overflows we extend both operands of size sz to $2 \cdot sz$ when building the division constraint, but only use the lower sz bits of the result further.

Signed division uses constraint eq. (14) as-is, but applies different restrictions on the remainder. First, the comparison from eq. (15) is performed on absolute values, which are calculated by taking the two's complement a value depending on its sign bit:

$$|rem| < |r| \tag{17}$$

$$|x| = \begin{cases} {\sim}x + 1 & \text{if } x[\text{size}(x) - 1] \\ x & \text{otherwise} \end{cases} \tag{18}$$

This however is still not sufficient as the following example shows, which would incorrectly accept $-1/2 = -1$:

$$\begin{aligned} &-1 = -1 \cdot 2 + 1 \\ &|1| < |2| \end{aligned} \tag{19}$$

To eliminate such cases, we restrict the remainder to be either 0 or negative if and only if the dividend is negative too, which corresponds for example to the definition of the % operator in C99 [3]:

$$(rem[\text{size}(rem) - 1] \iff l[\text{size}(l - 1)]) \lor \neg \bigwedge_i rem[i] \tag{20}$$

3.4 Incremental Flattening

As a consequence of the above constraint definitions, some operations such as multiplication and division produce significantly larger propositional formulas than others and are thus harder to decide for the SAT solver. In particular, consider the following example:

$$(a \cdot b = c) \land (b \cdot a = c) \land (x < y) \land (y < x) \tag{21}$$

It is apparent from only $(x < y) \land (y < x)$ that this formula is unsatisfiable. However, due to the complexity of the contained multiplications, our naive flattening implementation using minisat as the SAT solver, running on Arch Linux with an Intel Core i7 4790k CPU, only terminates with "Unsatisfiable" after approximately 15 seconds when deciding this formula on 32-bit vectors and does not terminate in a reasonable amount of time, after consuming the entire 16GB of memory of the test machine, when deciding on 64-bit vectors.

One approach in such cases is to solve the formula *incrementally*. The idea is to overapproximate the solution space by accounting for only a subset of constraints and thus deciding a more simplistic formula. If such an overapproximation is unsatisfiable, we can automatically conclude that the full formula is as well. Otherwise, if there are no constraints that the just found solution of the overapproximation does not satisfy, we have also found a solution for the full problem. If there are unsatisfied constraints, we add some of them to our approximation and repeat the procedure:

```
--| Arguments: constraints to be used now, constraints to be used later
incrementalSAT :: [Propositional.Formula] -> [Propositional.Formula]
                  -> Propositional.SolveResult
incrementalSAT current pending =
    case SAT.MiniSat.solve current of
        Unsat -> Unsat -- partial formula unsatisfiable ⇒ full formula unsatisfiable
        Solution assignment -> -- partial formula satisfiable
            let conflicts = -- all constraints that are False under the found assignment
                filter (λconstraint -> ¬ eval constraint assignment) pending in
            if conflicts == [] then
                Solution assignment -- no conflicts, full formula satisfied!
            else
                -- got conflicts, move the "easiest" from pending to current
                let new = conflicts[0] in
                incrementalSAT (current + new) (pending - new) -- continue
```

Since the constraints to be added can be freely chosen, we try to add more heavyweight constraints as late as possible in the iteration to avoid deciding them if not strictly necessary. In our implementation, we use a very primitive cost function summing up tokens of a propositional formula. Using it, the above incremental SAT solver can be brought into the context of bit vectors. We start with only the skeleton and add constraints in the order given by the cost function:

```
costEstimate :: Propositional.Formula -> Word

solveFlattenedIncremental :: FlattenedFormula -> SolveResult
solveFlattenedIncremental flat =
    let initialFormulas = [skeletonOf flat]
        incrementalFormulas = -- all formulas in the order in which they should be added
            sortOn costEstimate (allConstraints flat)
        satSolution = incrementalSAT initialFormulas incrementalFormulas
    in reconstructResult satSolution
```

Using this approach, our solver concludes the above example to be unsatisfiable instantly, or more specifically, the time taken is too insignificant to be directly compared against the previous runs.

4 Solving a CrackMe

Finally we want to present a practical application of the implemented solver. Given is a CrackMe, as commonly found in "Capture The Flag" contests, which is a small program available only as a compiled x86_64 executable for Linux. When run, it asks for a password to be entered on stdin and answers whether the input was correct:

```
$ ./crackme
Enter password: 123456
Wrong!
```

The challenge here is to find out the correct password by analyzing the binary without modifying it. This reverse engineering process, which would commonly be done using programs such as Radare2 [8] or Cutter [2], will be omitted here because it is out of scope of this paper. Instead, we directly work with C code that is equivalent to the contents of the binary, given in appendix A. The relevant part of this code is the function `check(char *s)`:

```
int check(char *s)
{
    for(size_t i = 0; i < 8; i++)
    {
        uint8_t a = s[i];
        uint8_t b = s[(i + 1) % 8];
        uint8_t c = s[(i + 2) % 8];
        uint8_t d = s[(i + 3) % 8];

        if(a & 0x80)
            return 0;

        uint8_t v = a << ((b + c) & 0b111);
        v = v | (a >> (8 - ((b - c) & 0b111)));
        v -= d;
        if(v != hash[i])
            return 0;
    }
    return 1;
}
```

This function accepts a string of length 8 and returns 1 iff the password is considered correct. The check itself consists of multiple bit vector arithmetic operations on the input characters, resulting in an 8-byte hash value. Trying to manually infer the expected input string from this is far from trivial, but it illustrates a perfect example application for our solver. We can write a small script, given in appendix B, running our solver on the following definition of a formula:

```
formula :: Formula
formula =
    conjunction $
        map (\i ->
            let a = pwChar $ i
                b = pwChar $ (i + 1) `rem` 8
                c = pwChar $ (i + 2) `rem` 8
                d = pwChar $ (i + 3) `rem` 8
                shiftPlus = Slice Unsigned 0 3 (b :+: c)
                shiftMinus = Slice Unsigned 0 3 (b :-: c)
                eight = Const Unsigned $ BV.pack [False, False, False]
                term = ((a :<<: shiftPlus) :|: (a :>>: (eight :-: shiftMinus))) :-: d
            in (Atom $ term :==: uConst (hash!!i)) :&&: (Not $ Atom $ Pick 7 a)
        ) [0..7]
```

While it may appear different in terms of syntax, it does in fact exactly represent the semantics of the above transformations done in C. Running the script yields the following unique solution:

```
$ stack runghc solve.hs
deadwing
```

And indeed, this is the correct password that the program accepts:

```
$ ./crackme
Enter password: deadwing
Correct!
```

5 Further Extensions and Applications

The presented operations are only one way of building a calculus on bit vectors. Extensions that can be decided with flattening include for example fixed point and floating point arithmetic. To optimize for the size of flattened formulas and performance, several preprocessing steps can be performed, such as replacing a division by a constant power of 2 by a static bit shift, resulting in a significantly simpler propositional formula. Furthermore, the explicit flattening of operators can be replaced by abstract uninterpreted functions as outlined in [4], which in particular helps deciding on equality between formulas.

The concept of flattening is also used in the well-known Z3 solver under the more general term *internalization* as described in [7] and implemented in [10]. Such solvers are used heavily in symbolic execution engines such as KLEE [1] or angr [9], which automate the process done manually in section 4 by executing code where certain memory regions are marked as symbolic, i.e. abstract and without a concrete value, and then using SMT to determine different concrete assignments for it which will make the program explore different control flow paths.

References

[1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[2] *Cutter*. URL: `https://github.com/radareorg/cutter` (visited on 06/03/2020).

[3] *ISO C Standard 1999*. Tech. rep. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[4] Daniel Kroening and Ofer Strichman. "Bit Vectors". In: *Decision procedures*. Springer, 2016. Chap. 6.

[5] *minisat*. URL: `http://minisat.se` (visited on 05/27/2020).

[6] *minisat-solver*. Hackage. URL: `https://hackage.haskell.org/package/minisat-solver` (visited on 05/27/2020).

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3." In: *LPAR Workshops*. Vol. 418. Doha, Qatar. 2008, pp. 123–132.

[8] *Radare2*. URL: `https://github.com/radareorg/radare2` (visited on 06/03/2020).

[9] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[10] *Z3, theory_bv.cpp*. URL: `https://github.com/Z3Prover/z3/blob/master/src/smt/theory_bv.cpp` (visited on 06/05/2020).

A CrackMe C Code

```c
#include <string.h>
#include <stdio.h>
#include <stdint.h>

uint8_t hash[8] = { 0xa2, 0x35, 0xa3, 0x0f, 0x1c, 0xd0, 0x0e, 0x9e };

int check(char *s)
{
    for(size_t i = 0; i < 8; i++)
    {
        uint8_t a = s[i];
        uint8_t b = s[(i + 1) % 8];
        uint8_t c = s[(i + 2) % 8];
        uint8_t d = s[(i + 3) % 8];

        if(a & 0x80)
            return 0;

        uint8_t v = a << ((b + c) & 0b111);
        v = v | (a >> (8 - ((b - c) & 0b111)));
        v -= d;
        if(v != hash[i])
            return 0;
    }
    return 1;
}

int main()
{
    printf("Enter␣password:␣");
    char str[0x10];
    fgets(str, sizeof(str), stdin);
    if(strlen(str) == 8 && check(str))
        printf("Correct!\n");
    else
        printf("Wrong!\n");
    return 0;
}
```

B CrackMe Solver Script

```haskell
import Common
import Formula
import Solve
import qualified BitVectorValue as BV

import Data.Maybe
import Data.Word
import qualified Data.Map as Map
import qualified Data.ByteString as B

import Control.Monad

hash :: [Word8]
hash = [0xa2, 0x35, 0xa3, 0x0f, 0x1c, 0xd0, 0x0e, 0x9e]

-- |Combine a list of formulas into a single formula one using And
conjunction :: [Formula] -> Formula
conjunction [a]        = a
conjunction (a : as)   = And a (conjunction as)
conjunction _          = undefined

-- |Variable name for the i-th character in the password
pwCharVarName :: Int -> String
pwCharVarName i = "pw[" ++ show i ++ "]"

-- |Term for the i-th character in the password
pwChar :: Int -> Term
pwChar = uVar 8 . pwCharVarName

formula :: Formula
formula =
    conjunction $
        map (\i ->
            let a = pwChar $ i
                b = pwChar $ (i + 1) `rem` 8
                c = pwChar $ (i + 2) `rem` 8
                d = pwChar $ (i + 3) `rem` 8
                shiftPlus = Slice Unsigned 0 3 (b :+: c)
                shiftMinus = Slice Unsigned 0 3 (b :-: c)
                eight = Const Unsigned $ BV.pack [False, False, False] -- Only 3 bits,
    overflow on purpose for 8 - x
                term = ((a :<<: shiftPlus) :|: (a :>>: (eight :-: shiftMinus))) :-: d
            in (Atom $ term :==: uConst (hash!!i)) :&&: (Not $ Atom $ Pick 7 a)
        ) [0..7]

main :: IO ()
main =
    case solveAll formula of
    Solution s ->
        forM_ s (\s ->
            putStrLn $ toEnum <$> fromIntegral <$> map (\i ->
                               let bv = s Map.! (pwCharVarName i) in
                               (fromJust (BV.fromBitVector bv))::Word8) [0..7]
            )
    res -> print res
```