Using Coq to Verify DPLL

Kenneth Roe http://www.cs.jhu.edu/~roe The Johns Hopkins University

Summary

Formal Methods for Imperative Languages Such as C

Many bugs such as Heartbleed relate to violations of data structure invarants

The object of our research is to create tools for documenting and reasoning about complex data structure invariants

DPLL algorithm is a well suited example
- Data structure has very complex invariant
- Implementation code is small

Research Contributions

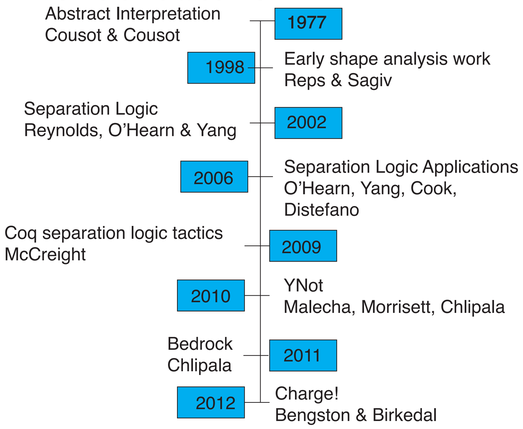
Extension of Coq separation logic reasoning to larger programs with more complex data structures.

Creation of a library of many useful tactics (such as a powerful simplification tactic)

Better understanding of the types of proofs that need to be done for complex invariants

Creation of a Python Coq based IDE (with scripting capabilities) to automate many common theorem proving tasks

Timeline of related work:



Preventing Heartbleeds

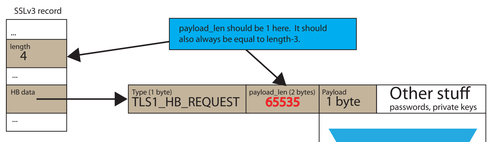
Logo from heartbleed.com

Heartbleed is a bug in OpenSSL code
- Thousands of websites use OpenSSL for HTTPS
- Websites vulnerable to sensitive data stealing attacks

Bug is in newly added heartbeat code
- A heart beat is sent once every few seconds by one side of an SSL connection to check if the other side is alive
- The other side responds with a message echoing the payload data

Exploitation of this bug is shown below
- Message has broken payload_len field
- Response constructed based on payload_len field
- No bounds check on memcpy
- Data beyond end of allocated block copied over

Attacker request



Victim Response

The theorem proving techniques presented on this poster to verify DPLL could also be applied to OpenSSL and would have almost certainly found the Heartbleed bug. We suggest two approaches as to how our techniques could be used to block Heartbleed.

1) Since packets being received cannot be trusted, add sanity checks to verify the invariants. Formal methods could be used to verify that the sanity checks work and to verify that once the invariants are satisfied, that unauthorized information cannot leak out
2) Separation logic can be used to add a pre-condition to memcpy that insures that it does not copy beyond the end of the record which the source pointer references. An unchecked parameter to memcpy was key to the Heartbleed bug. Formal methods could verify that the parameters passed to memcpy are sound.

The basics: a traversal program

Proof steps involve the following:
(1) forward propagating over unit statements
(2) applying Hoare rules to break up if-then-else, while and statement sequences
(3) folding or unfolding recursive predicates
(4) merging two states at the end of an if-then-else
(5) proving one state entails another state

Most of these steps are simple and heuristics exist to apply them automatically. Sometimes, however, merge or entailment can become quite complex and a large proof may be required. These few complex steps represent 99% of the verification effort needed.

Pre-condition

```
(*{{AbsExistsT v0 .
   (AbsExistsT v1.
     (AbsExistsT v2.
       (TREE(R, v0, 2, [0,1]) *
        TREE(I, v1, 2, [0]) *
        TREE(P, v2, 2, [0]) *
        AbsAll v3 in TreeRecords(v1). ([nth(find(v1, v3), 2) inTree v0]) *
        AbsAll v3 in TreeRecords(v2). ([nth(find(v2, v3), 2) inTree v0]) *
        [T == 0 \/ T inTree v0])))}}
```

Program

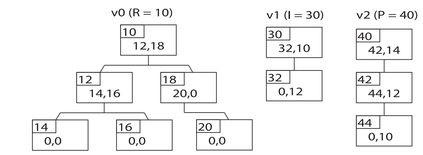
```
WHILE not (T == 0)
DO N := P;
  NEW P, 2;
  *(P + 1) := T;
  *P := N;
  Tmp_I := *T;
  Tmp_r := *(T + 1);
  IF (Tmp_I == 0) and (Tmp_r == 0)
  THEN IF I == 0 THEN T ::= 0
       ELSE T := *(I + 1);
            Tmp_I := *(I + A0); DELETE I, 2; I ::= Tmp_I
  ELSE IF Tmp_I == 0 THEN T ::= *(T + 1)
       ELSE IF Tmp_r == A0 THEN T ::= *T
            ELSE N := I;
                 I := NEW 2;
                 *(I + 0) := N;
                 Tmp_I := *(T + 1);
                 *(I + 1) := (!Tmp_I); T ::= *T LOOP
```

Post-condition

```
{{AbsExistsT v0 .
  (AbsExistsT v1.
    (AbsExistsT v2.
      (TREE(R, v0, 2, [0,1]) *
       TREE(I, v1, 2, [0]) *
       TREE(P, v2, 2, [0]) *
       AbsAll v3 in TreeRecords(v2), ([nth(find(v2, v3), 2) inTree v0]) *
       [T == 0)))))*}
```

Coq Verification goal

Heap snapshot



Invariant

(1) There is a tree and two lists all properly formed--no cycles, no dangling points
AbsExistsv0.AbsExistsv1.AbsExistsv2.
TREE(R, v0,2,[0,1])*TREE(I, v1,2,[0])*TREE(P, v2,2,[0])*
(2) Each of the nodes in the two lists contains a pointer a node in the tree.
AbsAll v3 in TreeRecords(v1).
[nth(find(v1, v3),2)inTreev0]*
AbsAll v3 in TreeRecords(v2).
[nth(find(v2, v3),2)inTreev0]*
[T=0\/T inTreev0]

Functional Equivalent Representation

Characterize data fields saved, structure of the tree or list and the locations of records on the heap.

Represented by lists whose elements are either naturals or (recursively) lists of lists or naturals.

For the snapshot from the picture above, v0 would for example be the list
[10,([12,[14,[0],[0]],[16,[0],[0]]]),([18,[20,[0],[0]],[0]])]

Verifying the DPLL algorithm

Efficient SAT solving algorithm for CNF expressions such as:

(A v ~B v C) ^
(A v ~C v D)

DPLL algorithm:
(1) Choose a variable and assign it a value
(2) Perform unit propagation to find additional assignments implied by the choices already made.
(3) Backtrack and change choices when a contradiction is found

Watch variables
(1) Makes unit propagation very efficient
(2) Two unassigned variables chosen at random
(3) If a watch variable in a clause is assigned, then choose a replacement. If one cannot be found, then there is only one assigned variable left and a unit propagation needs to be performed.

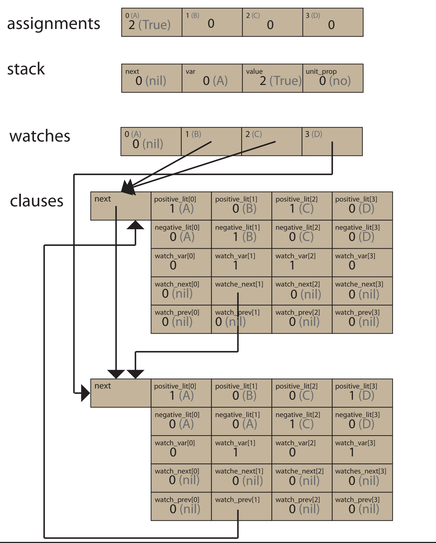
Blue variables in the example above are initial watch variables After A is assigned false, we move the watch that was on A in the two clauses to C and D respectively (the brown variables).

Our C program uses the following data structures to store the clauses, the watch variable linked lists, the assignments and a "todo" queue.:

```
#define VAR_COUNT 4

char assignments[VAR_COUNT];

struct clause {
  struct clause *next;
  char positive_lit[VAR_COUNT];
  char negative_lit[VAR_COUNT];
  char watch_var[VAR_COUNT];
  struct clause *watch_next[VAR_COUNT];
  struct clause *watch_prev[VAR_COUNT];
} *clauses;

struct clause *watches[VAR_COUNT];

struct assignments_to_do {
  struct assignments_to_do *next, *prev;
  int var;
  char value;
  int unit_prop;
} *assignments_to_do_head, *assignments_to_do_tail;

struct assignment_stack {
  struct assignment_stack *next;
  int var;
  char value;
  char unit_prop;
} *stack;
```

Here is a diagram showing what the data structures look like right after A is assigned false.

assignments_to_do_head=nil
assignments_to_do_tail=nil



Coq data structure invariant

Contains all of the important properties in about 50 lines of Coq code. A fragment of the invariant is shown in the box below. Here is an informal statement of the watch variable invariant:

All clauses have two watch variables. For each clause, one of the following three cases is true:

1) The two watch variables are unassigned
2) All but one variable is assigned in the clause. One of the watch variables is the unassigned variable. The other is the most recently assigned variable
3) At least one of the assignments satisfies the clause. If one or both watch variables are assigned, then those assignments were either a satisfying assignment or done after the first satisfying assignment.

Consider this piece of code that removes the most recent assignment:

```
var = stack->var;
value = stack->value;
struct stack *n = stack->next;
free(stack);
stack = n;
assignments[var] = 0;
```

This code removes the most recent assignment. Proving that the invariant above is correct for each clause involved the following cases for each clause:

1) Two watch variables are assigned before
2) All but one variable is assigned but the assignment removed does not appear in the clause
3) All but one variable is assigned and the assignemnt removed does appear in the clause
4) At least one of the assignments satisfies the clause. The one and only satisfying assignment is the variable being removed
5) At least one of the assignments satisfies the clause. The one assignment removed is not a satisfying assignment.
6) At least two of the assignments satisfies the clause. The one assignment removed is a satisfying assignment.

The proof that the invariant holds after this code took over 2000 lines of Coq proof script code.

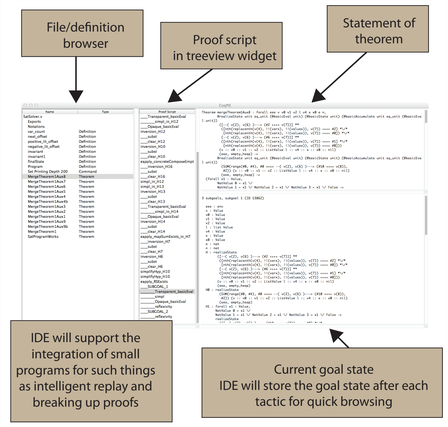
```
The first part of the invariant are spacial constructs asserting the two arrays and
three dynamic data structures in the heap. ARRAY(root, count, functional_representation)
is a spatial predicate for arrays. The functional representation is a list of the
elements.

AbsExistsT v0 , AbsExistsT v1 , AbsExists v2 , AbsExistsT v3. AbsExistsT v4.
      TREE(clauses,v0,sizeof_clause,[next_offset]) *
      TREE(assignments_to_do_head,v1,sizeof_assignment_stack,[next_offset]) *
      TREE(stack,v2,sizeof_assignment_stack,[next_offset]) *
      ARRAY(assignments,var_count,v3) * ARRAY(watches,var_count,v4) *

Next, we add on two assertions that guarantee that both the assignment_stack v2
and assignment array v3 are consistent. We use (a,b)--->c as an abbreviation for
nth(find(a,b),c).

      (AbsAll v5 in TreeRecords(v2) . ([nth(v3,(v2,v5)--->stack_var_offset)==(v2,v5)--->stack_val_offset)]) *
      (AbsAll v5 in range(0,var_count-1) . ([nth(v3,v5)==0] "\/" AbsExists v6 in (TreeRecords(v2)) .
            ((((v2,v6)--->stack_var_offset==v5 /\ (v2,v6)--->stack_val_offset==nth(v3,v5))))) ) *

The TREE declarations above define linked lists. They do not define the back
pointers of a double linked list to do this for the assignments_to_do_head list, we
add the following assertion:

      (AbsAll v5 in TreeRecords(v1) .
         ((((v5,v1)--->prev_offset==0 /\ (assignments_to_do_head)==v5) \/
          ((v5,v1)--->prev_offset inTree v1 /\ (v5,(v5,v1)--->prev_offset)--->next_offset==v5)))) *

Now we define the linked lists for each of the watch variables. Path is like TREE
but it defines lists inside of other structures. It is used for the embedded watch
variable linked lists. It takes the form:
Path(root, parent_functional_data,child_functional_form,node_size,pointer_offsets)
We also put in the assertion for the prev links:

      (AbsAll v6 in range(0,var_count-1)
         (Path((nth(v4,v5)), v0, v6, sizeof_clause, [watch_next_offset+v5]) *
          (AbsAll v7 in TreeRecords(v6)
             ((((v6,v7)--->(watch_prev_offset+v5)==0 /\ nth(v4,v5)====v6) \/
              ((v6,(v6,v7)--->(watch_prev_offset+v5))--->(watch_next_offset+v5))==v7))) ) *
...
```

Coq DPLL invariant fragment

Statistics:

Code size:	~200 lines
Invariant size:	~52 lines
Size of verification so far:	~3700 lines
Estimated size of complete verification:	~20,000 lines

In progress: Python IDE



Developing complex proofs using Coq is quite tedious. While developing our proofs, we ran into the following issues:

1) Often while proving a lemma, we find that the statement of our invariant is not correct. If many other lemmas have already been completed, when we change our invariant, all of our older lemmas need to be updated. These updates may be non-trivial has Coq generated hypothesis names may change due to a change in the number of hypotheses.
2) Coq will slow down substantially (or even crash) when the proof of a single lemma becomes very large or if there are a large number of hypotheses. (Applying a single tactic can take over a minute)
3) Coq provides the Ltac langauge for creating scripts to automate theorem proving tasks. However, Ltac is missing many basic operators. Many tasks which should be easy to automate cannot be expressed in Ltac. For example, one cannot easily test if a subterm is a variable or something more complex.
4) Often one needs to peek at an earliler state while working on a proof script. The cost of cancelling and then replaying a portion of the script can often be quite expensive.

Conclusions

The techniques presented on this poster will lead to an effective tool for debugging software.
- Data structure invariants effective in finding critical bugs
- DPLL good test case for developing data structure invariant verification techniques
- Biggest challenge is dealing with productivity issues in developing correctness proofs.
- Without an IDE, we guesstimate it takes one hundred days of verification for each day of development.

Future work

- Test framework on larger programs such as OpenSSL
- Completion/refinement of framework for larger verifications
- Python IDE needs to be completed.
- Function calls/programs with more than one procedure
  - Separation logic and the frame rule gives a good basis
- Concurrency
  - Still an open area

Acknowledgements: The author would like to thank Scott Smith, for his feedback on this poster.