Verifying programs with Complex data structures using Coq

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

Motivation

95% of software bugs such as Heartbleed relate to violations of data structure invariants

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

Proof development productivity is the key issue that needs to be addressed.

One can build up the data structure invariants starting with an attempt to prove that a program does not have certain types of bad behavior such as:
- nil pointer references
- dangling pointer references
- array out of bound references
- unallocated memory reads/writes

As an example, the Heartbleed bug could have been found just from a specification stating that there were no reads of unallocated memory

To study data structure invariant verification, we chose to verify a simplified version of DPLL

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

We implemented a variation that is about 200 lines of C code, supports the 2 watch variable unit propagation algorithm but not learning.

We then developed an invariant for the data structures which includes all the dependencies related to mainting the two watch variables.

We are working on a verification of this invariant.

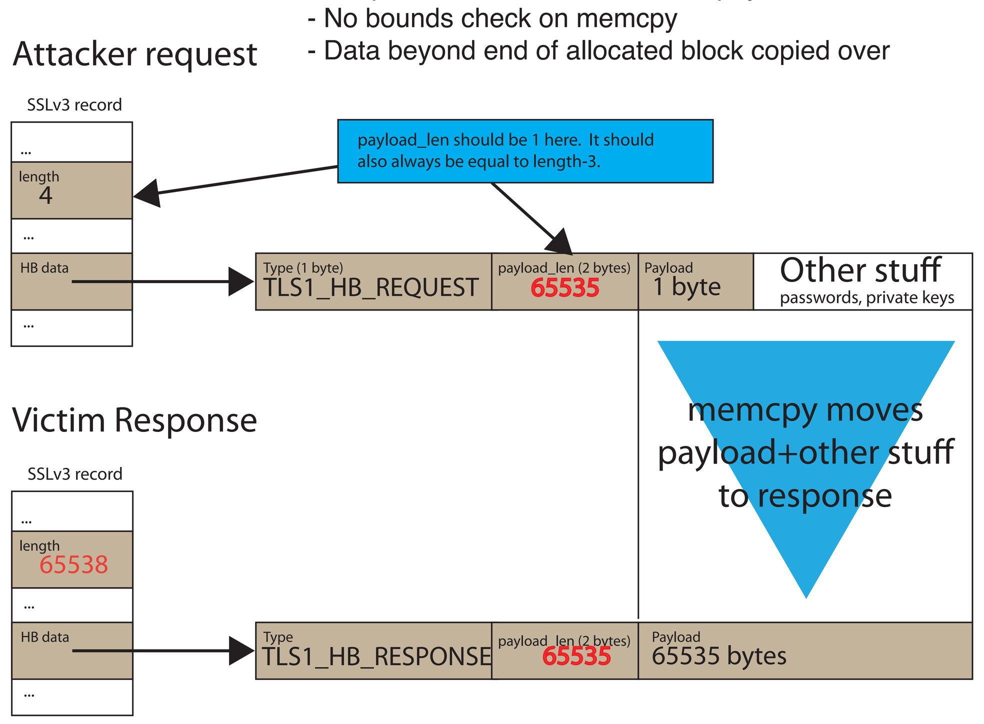
Preventing Heartbleeds

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



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 to memcpy are sound.

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

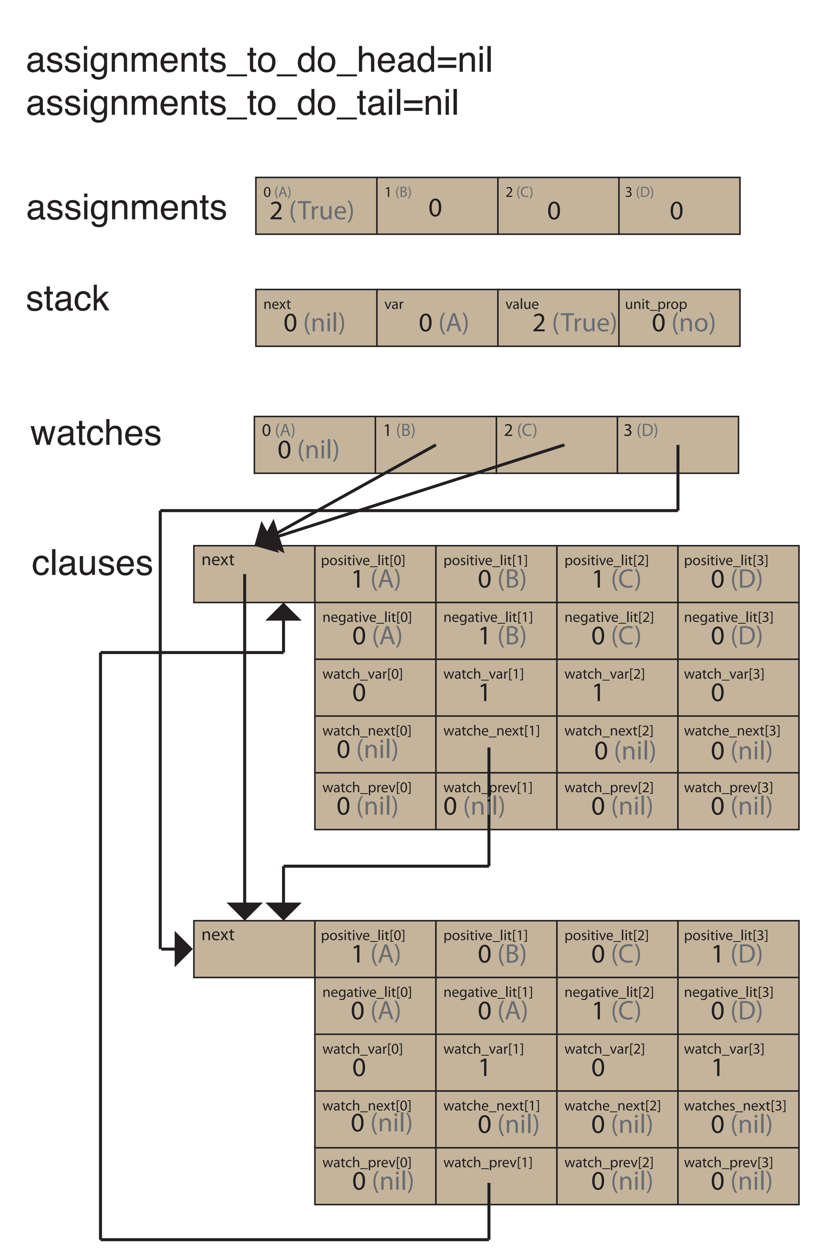
Watch variables
(1) Makes unit propagation very efficient
(2) Two unassigned variables chosen at random
(3) If a watch variable in a clause is assigned, then choose a replacement. If one cannot be found, then there is only one assigned variable left and a unit propagation needs to be performed.

Blue variables in the example above are initial watch variables After A is assigned false, we move the watch variable 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.



The Challenge of Prover Productivity

* Prover productivity ratio

Time to verify code
Time to develop code

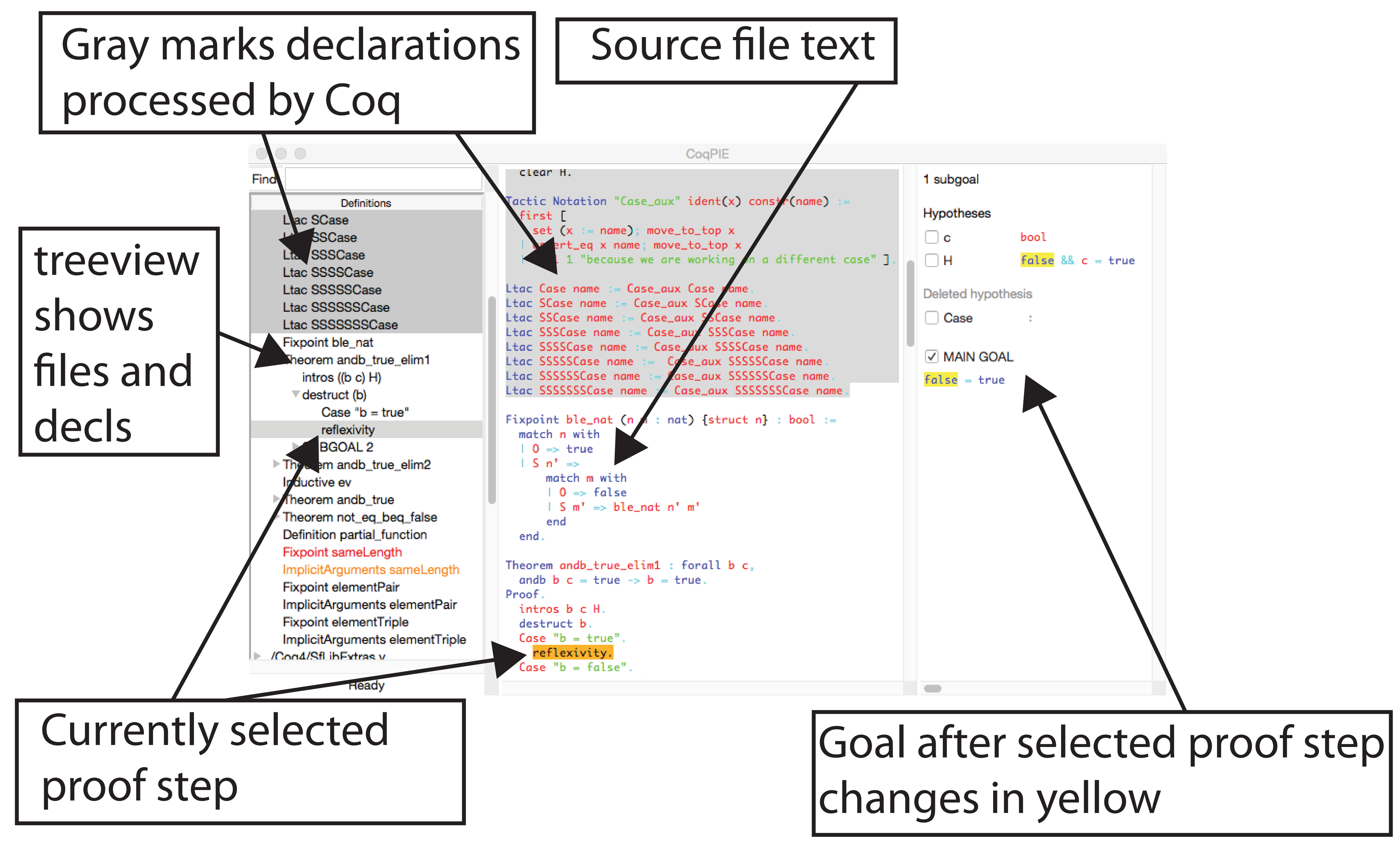
* Currently, this ratio is well over 100/1 for any interactive theorem based verification
* At 10/1, a fairly valuable software development tool could be produced

CoqPIE: Improving proof development productivity

Our strategy for improving proof development productivity is to introduce a new IDE addressing these issues:

(1) One often finds errors in the statement of a theorem while developing a proof. Changing the statement often require the proof script to be updated
(2) Often it is useful to quickly review earlier goal/hypothesis states. Existing IDEs require the Coq prover itself to backup or move forward in a script. On a complex proof each step can take 30 seconds to backup or reevaluate.
(3) LTac--the scripting langauge for Coq has many holes. Many simple rules cannot be expressed
(4) Navigating to specific definition declarations can be difficult if there is a large amount of proof script code.

CoqPIE is about 13000 lines of Python source code
Windowing implemented with TkInter



Key Features of CoqPIE

* CoqPIE saves Coq output after each proof step
* CoqPIE maintains an AST parse tree of the source code and Coq output
* AST incrementally updated as edits are made
* The relationship between AST nodes and source text is maintained.
* Complex tactics such as replay and lemma extraction built on top of AST representation
* CoqPIE manages the entire project--not just one file
* Treeview shows summary of all files and definitions
* Definitions and proofs that need to be recompiled due to changes to dependent definitions are highlighted
* Diference highlighting allows one to quickly see changes after each proof step

Dependency information

When a definition or lemma is edited, it can impact the validity of other declarations that depend on it.
* CoqPIE automatically marks declarations that have been invalidated.

Mitigating Coq performance

One of the biggest sources of productivity problems is the speed of the Coq theorem prover. Complex proofs can take hours (or even days) to fully verify. A single step can take a minute to process in a long proof.
(1) All intermediate goals are cached. Simply reviewing a proof does not invoke Coq.
(2) CoqPIE provides tactics to break up large proofs into lemmas--this often improves the performance of Coq
(3) CoqPIE will replace a proof script with admit if you are simply jumping over an entire theorem

Replay

Often the process of developing a theorem reveals errors in the statement of a theorem. When that statement is changed, the script needs to be adapted. Changes may involve the following:
(1) Removing proofs for subgoals that vanish
(2) Creating stubs for new subgoals
Often the error discovered in a theorem declaration is a missing antecedant
(3) Updating hypothesis names in tactics. Adding an antecedant may shift down hypothesis numbers. eg. "inversion H10" may need to become "inversion H11"
Replay works by having both the old and new output at each step. CoqPIE can then analyze differences to find what needs to change in the tactic.

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