Achieving High Speed and High Assurance in a Hardware-Based Cross-Domain System using Guardol

David Hardin
Konrad Slind
Collaborators

• Mark Bortz — Hardware expert
• Doug Hiratzka, Jim Potts — Embedded systems software experts
• Mike Whalen, Hung Pham (University of Minnesota) — RADA solver for algebraic data types
• Scott Owens — unpaid consultant
Motivation

• As critical systems become networked, they become vulnerable to cyber attacks

• New cyber vulnerabilities appear regularly: Shellshock, POODLE, Heartbleed, etc.
  • Appearance of vulnerabilities has outpaced industry’s ability to find and fix
  • A number of these flaws have been present for years, and many have survived the scrutiny of the "many eyes" of open source development

• Critical systems tend to use the same operating systems, network stacks, etc., as commercial off-the-shelf-systems
  • But, older “stable” versions tend to be used, making critical systems vulnerable to known attacks

• Critical systems are not patched often, leading to long exposure times
Cross-Domain Systems (CDS, a.k.a. Guards)

- A guard mediates information flow between security domains according to a specified policy.
- Guards are often implemented on top of some “high-assurance” operating system, but usually not the current release of that OS
  - Very long exposure time for vulnerabilities
  - Often, the Operating System is just an old version of Linux
- The guard policy is generally a rule set that is interpreted on a packet-by-packet basis by the guard software
  - The language used to encode guard logic is peculiar to the individual guard vendor
  - Little to no automated V&V support
  - Performance is highly variable, depending on rule complexity

The Guardol program is significantly advancing the state of the art in guard portability, assurance, and performance.
Rockwell Collins CDS Products

- Turnstile
  - AAMP7G-based
  - DCID 6/3 PL5

- MicroTurnstile
  - AAMP7G-based
  - Very low power, wearable
  - “Bump in the wire” USB Guard

- SecureOne Guard
  - Based on commercial separation kernel technology
  - Shares Guard Engine software with Turnstile and MicroTurnstile

- US Patents 7,606,254, 8,161,259, and 8,881,260
Guardol Objectives

- Develop a *Domain-Specific Language* (DSL) for guards
  - A DSL is a programming language dedicated to a particular problem domain, representation style, and/or solution technique
- Automate the design flow
  - Analysis and implementation artifacts automatically generated with high assurance
- Integrated analysis capability
  - Formalization of a Guardol source program automatically generated by the frontend of the Guardol toolchain
  - Middle-end of Guardol utilizes the HOL4 theorem prover, operating in “headless” fashion
  - Model checking of key requirements from the guard specification
- Support for a wide variety of guard platforms
  - Demonstrated operation on a competitor’s guard
  - Demonstrated real-time imaging guard for MicroTurnstile
  - Able to support high-performance hardware guards (New work)
Guardol Eclipse Environment

debug mode

type $ = string;

$ = string;

$ = string;

$ = string;

$ = string;

-- Imports

imported function

msgPolicy(Text : in $, Output : out $);

-- The guard

function Guard (Input : in $, Output : out $) =

begin

var

ValueResult : $;
LeftResult : $;
RightResult : $;

in

match Input with

TreeLeaf => Output := TreeResult'OK(TreeLeaf);
TreeNode node =>

begin


The Guardol Language

- Guardol can be characterized as a “mashup” of concepts from Ada, ML, and the C family of languages.
- Guardol is, first and foremost, a strongly-typed imperative language, with assignment, functions, for loops, while loops, etc.

```plaintext
function guard_main() = {
  var sz: int;
  pkt: GMTI_Pkt;
  in
  while (sz > 0) do {
    guard_result := guard(pkt);
    ...
  }
}
```
The Guardol Language (cont’d.)

• Types, however, are influenced by the functional language ML:

```plaintext
type Tree = [elem: int, rank: int, children TreeList]

type TreeList = { Nil
                   | Cons : [hd: Tree, tl: TreeList]}

type IntOpt = { NONE | SOME : int }
```
The Guardol Language (cont’d.)

- Guardol also inherits the powerful “match” operator from ML:

```plaintext
function ins (t: in Tree, tlist: in TreeList) returns Output: TreeList = {
    match tlist {
        'Nil => Output := 'Cons[hd: t, tl: 'Nil] ;
        'Cons c =>
            if t.rank < c.hd.rank then
                Output := 'Cons [hd: t, tl: tlist];
            else
                Output := ins(link(t, c.hd), c.tl);
        }
    }
}
The Guardol Language (cont’d.)

- Many of the design decisions in Guardol anticipated features of “hot” new programming languages, e.g. Apple’s Swift:

```swift
static func ins (t: Tree, tlist: TreeList) -> TreeList {
    switch tlist {
    case .Nil: return TreeList.Cons(t, TreeList.Nil);
    case let .Cons(hd, tl):
        if t.rank < hd.rank {
            return TreeList.Cons(t, tlist);
        } else {
            return ins(link(t, t2: hd), tlist: tl);
        }
    }
}
```
Guardol Property Specifications and Proofs

- A novel Guardol feature is the ability to state and prove formal property specifications directly in the source text, using Guardol language syntax
- The following property spec conjectures that if a TreeList is rank-ordered, it is still rank-ordered after a new tree is inserted:

```plaintext
spec rank_ordered_ins = {
    var t: Tree;
    list: TreeList;
    in
        if rank_ordered(list)
            then check rank_ordered(ins(t, tlist));
        else skip;
}
```

- The Guardol verification backend proves this property automatically
Adding Regular Expressions to Guardol

- Based on customer feedback, we have recently added regular expression support to Guardol.

- Regular expression matching can be invoked within a Guardol program by:

  ```verdict := regex_match(rlit, s);```

  where `rlit` is a regular expression literal, `s` is a string, and `verdict` is a boolean result.
Guardol Regular Expression Literals

- Regular expression literals in Guardol largely conform to the syntax found in languages like Python.

\d = 0..9
\w = [a-zA-Z0-9_]  
. = any char except \n
\s = whitespace = [ \n\r\t\f\] (* Note the space character! *) \t = tab
\n = newline
\r = return
\f = formfeed
\c = escape c
rs = concatenation r | s = disjunction r* = Kleene star
r+ = rr*
r? = "" | r
r{n} = r^n
r{m,n} = r{m} | r{m+1} | ... | r{n} (m<=n) r{m,} = r{m}r*
r{,n} = r{0,n}
(r) = grouping
[... ] = character set
Fast Regular Expression Matching

- Brzozowski (1964) presents a method for compiling a regular expression to a Deterministic Finite-state Automaton (DFA), which is subsequently run on strings.

- The essential insight behind Brz is that regexes are identified with DFA states:
  - The given regex $r_0$ is the start state
  - For each symbol $a_i$ in the alphabet, compute $r_a = \text{Deriv } a_i \ r$. The $r_a$ are the successor states to $r$
  - Stop when no new states are created
  - Final states are those that match the empty string

- Thus, regular expression matching becomes very fast
Compiling Regular Expressions to DFAs

- The following pseudo-code executes DFA $d$ on input $s$:

  ```plaintext
  Exec_DFA (d:DFA, s:string) returns verdict:boolean = {
    var
      q, len : int;
    in
      len := s’Length;
      q := d.init;
      for (i=0; i<len; i++) { q := d.trans[q,s[i]]; } 
        verdict := member(q,d.final);
  }
  ```

- Brzozowski provided a pencil-and-paper proof of the correctness of his DFA compilation approach
- We have crafted a formalization in HOL4, which we utilize in Guardol code generation
  - Employs a counter to avoid tricky termination issues
Side Note: Code Generation vs. Proof

- A regex_match expression in a Guardol program is treated differently whether code is being generated, or properties are to be proved.

```plaintext
regex_match(rlit, s)  \rightarrow \text{Matches r s}
```

Property Generation

```plaintext
Code Generation
Exec_DFA(Brz(r), s)
```
Guardol Regular Expression Demo Program

package Regex =

-- Filter for full syslog message. Meant to handle messages conforming to either
-- RFC 5424 or RFC 3164. Skips over leading information by looking for an occurrence
-- of a space followed by an open bracket, i.e., " [". After that, it expects
-- the remainder of the structured data portion of the message.

function syslog_5424_or_3164_filter (input : in string) returns verdict : bool = {
    verdict := regex_match(
        \.* \[\{"time":\d{13}(:\d{3})?","w\{1,20\}":\{("w\{1,25\}":"w\{1,30\}",?)\}+\}\}\,
        input);
}

This filters a syslog packet against a JSON-based packet format.
package body Regex is

    function execDFA_1 (str : in String) return Boolean is
        verdict : Boolean; state : uint; len : uint; i : uint;
    begin
        state := Regex.DFA_1.start; i := 0; len := str'Length;

        while (i < len) loop
            i := (i + 1);
            state := Regex.DFA_1.trans(Natural(state),Natural(Character'Pos(str(i))));
        end loop;

        verdict := Regex.DFA_1.final(Natural(state));
        return(verdict);
    end;

    function syslog_5424_or_3164_filter (input : in String) return Boolean is
        verdict : Boolean;
    begin
        verdict := Regex.execDFA_1(input); return(verdict);
    end;
end Regex;
Guardol for Hardware Guards Program Overview

- Adapt an existing Rockwell Collins FPGA-based board with dual network interfaces to function as a guard
- Adapt the Guardol toolchain to generate VHDL for a subset of legal Guardol programs
- Demonstrate a Guardol regular-expression based guard running on the hardware
- Provide performance measurements for the hardware-based guard
- Produce guidance for the modification of accreditation artifacts for a Rockwell Collins product guard, such as the SecureOne Guard
Guardol for Hardware Guards (Analysis not shown)

Guardol Source

Guardol Frontend

VHDL source

VHDL Compiler

Bit file

CPU Core

“High” Network

“Low” Network

Guard RAM

Accept/Reject

Rd/Wr

Guardol-Derived Guard Logic

FPGA

© 2015 Rockwell Collins. All rights reserved.
Guard Hardware
Guardol-to-VHDL Code Generation

- Modify the Guardol code generator to produce VHDL

- VHDL and Ada are very similar syntactically, but differ significantly at the semantic level
  - Must always be concerned about parallelism

- Not all legal Guardol programs will be able to be translated to VHDL initially, e.g. Guardol programs that allocate memory
  - However, regular expression guards are readily translatable to VHDL

- Have performed preliminary translations and have successfully simulated them using the Xilinx tools; currently in synthesis
Ada vs. Synthesizable VHDL

- Easy to translate Ada to compilable, but non-synthesizable VHDL
  - Numerous small syntactic differences

- However, it’s not always clear whether a given VHDL model is synthesizable
  - Can fail along either time or space dimensions
  - Often, just have to try to synthesize, and see what happens

- Sequential execution model -> Parallel execution model

- Many other changes, often, but not always, needed
  - Variables -> signals
  - Boolean type -> std_logic
  - Integer type -> std_logic_vector
  - Loop, with control variables -> process, with sensitivity list
  - String -> RAM entity
architecture RTL of GUARDOL_GEN is

constant DFA_1 : DFA_1_components :=
  (trans => [...]   — array of next state values
  start => 1,      — start state
  final => [...]);  — boolean array indicating final states;

begin

  ACCEPT_REJECT: process (q_curr_state)
  begin
    if (DFA_1.final(Natural(q_curr_state))) then
      i_accept <= '1';  -- accept
      i_reject <= '0';
    else
      i_accept <= '0';  -- reject
      i_reject <= '1';
    end if;
  end process;

  STATE_MACHINE_DFA : process (q_curr_state, DATA_VALID, NEW_PACKET)
  begin
    if (NEW_PACKET = '1') then
      n_curr_state <= DFA_1.start;
    elsif (DATA_VALID = '1') then
      n_curr_state <= DFA_1.trans(Natural(q_curr_state), to_integer(unsigned(DATA)));
    else
      n_curr_state <= q_curr_state;
    end if;
  end process;

end architecture;
Performance

• Regex guard is designed to examine one byte per clock tick

• Guard RAM is 64 bits wide, so only need to read the RAM once every 8 bytes
  • Incur an additional one cycle delay in this case

• System clock for existing guard hardware is 167 MHz

• One extra clock at the end to latch the final accept/reject result

• Thus, the performance for a regular expression guard is approximately 1.2 Gbps
  • Our VHDL simulations support this result
Conclusion

Guardol technology enables the development of a new class of hardware-based guards with significantly higher assurance and greater performance.

**High Assurance:**
- Hardware guard engine for best protection against attacks
- Automated formal proofs of guard properties
- Formal proof of correctness for compilation for regex guards

**High Performance:**
- Guardol compilation to hardware much higher performance than rule interpretation in software
- Goal is to achieve line-speed operation for complex regular-expression guards