Introduction to the Guardol Language and Verification System

David Hardin
Trusted Systems Group
Rockwell Collins Advanced Technology Center

December 6, 2011

Colleagues

This work has been conducted along with

- Konrad Slind, Rockwell Collins
- Andrew Gacek, Rockwell Collins
- Mike Whalen, U. Minnesota
- Tuan-Hung Pham, U. Minnesota
- John Hatcliff, Kansas State U.
- Joey Dodds, Kansas State U.
- David Greve, Rockwell Collins

Part 1

Guards and their properties

What is a guard?

A **guard** is a device that mediates information sharing between security domains according to a specified policy.

Typical guard operations on a packet stream:

- read field values in a packet
- change fields in a packet
- transform packet by adding new fields
- drop fields from a packet
- construct audit messages
- remove entire packet from stream

Typical Guard Structure

A guard

- is hosted on some high-robustness operating system
    - Thus, a guard inherently constitutes a layered assurance problem
- is multi-homed (has network interfaces for the n networks it mediates)
- is generally (semi-)programmable via a system-specific set of rules
- has traditionally been applied to relatively simple packet types, but guards for tree-structured data of arbitrary size (*e.g.*, email, XML) are increasingly needed

Specific guard properties

What might we want to assert about a guard?

- The guard should be **NEAT** (Non-bypassable, Evaluatable, Always Invoked, Tamper-proof)
- The output packet has no occurrence of some field in the input packet
- No “dirty” words exist in the output packet
- No information that is not releasable to a particular destination is transmitted to that destination
- Target email addresses don’t contain `.rogueNation`
- Every field labelled `foo` in the output has been fuzzed, or encrypted

Guard technology at Rockwell Collins

Rockwell Collins has accumulated some experience in the area

- 2005: High assurance guard demo
- 2007: Turnstile
  - based on AAMP7 microprocessor
  - in production
- 2010: MicroTurnstile
  - used to guard USB comms in soldier systems
  - also AAMP7 based
  - size of a pack of gum
  - in final development

Some problems with guards

Guards can be used in a wide variety of settings (commercial, medical, military) so it is difficult to generalize, BUT

- Not a lot of literature (or information-sharing) on guards
- A guard is a {safety,privacy,mission}-critical system component which should be verified, but guard evaluation standards are currently in flux
- Portability is hardly addressed
- Performance of rule-based guards is difficult to assess

Uphsot

Guards can be slow to build and then to be certified.

Guards may be slow when executing.

There is little support for guard verification, or for exploring guard properties.

Our design

Our approach is to develop a **domain-specific language** for guards, plus support technology.

- Automatic generation of implementation and formal analysis artifacts
- Integrate and highly automate formal analysis
- Ability to glue together existing or mandated functionality
- Support a wide variety of guard platforms

Part 2

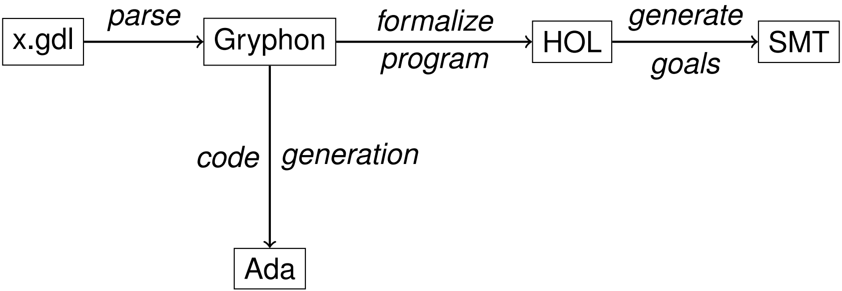
Guardol

The Guardol language

Roughly: **Guardol** $=$ **Ada** $+$ **ML**

- Ada provides a familiar setting (types, programming constructs) for our target programmers.
- ML datatypes succinctly capture tree-structured data, *e.g.*, email, XML.
- We placed relatively little emphasis on incorporating cutting edge programming language features.
- Guardol is intended to be a **fairly simple language with cutting edge verification support**.

The Guardol system



Guardol language summary

Guardol is a conventional imperative language with ML-style datatypes.

- standard base types (**bool**,**int**,**word32**,**string**)
- record types
- mutual, nested recursive types
- standard imperative programming constructs (assignments, procedures, sequential composition, *etc*)
- pattern-matching
- declarations for external functionality
- specification construct
- package system

What Guardol doesn't have

- **no** infinite loops
  - A guard should always complete its task. Also, proof automation for recursive programs is based on induction, which requires termination.
- **no** pointers
  - Pointers complicate reasoning. Guardol provides automatic memory management for unbounded tree-shaped structures when generating code.
- **no** I/O
  - Guardol is aimed at just the guard, not its computational context, *i.e.*, how data gets to it, and how its output is managed.
- **no** ML-style polymorphism (not yet, anyway)
  - All data structures are ground, *i.e.*, have no polymorphic types. This makes some aspects of processing easier, and is more familiar to some programmers.

Externals

One design goal of Guardol is to be able to use pre-existing functionality, provided by the platform, or when a particular implementation is mandated. Syntax of the declaration:

$\texttt{imported function}\ \textit{name}\ (\textit{arg}_1, \ldots, \textit{arg}_n);\quad \text{or}$

$\texttt{imported function}\ \textit{name}\ (\textit{arg}_1, \ldots, \textit{arg}_n)\ \texttt{returns}\ \textit{name} : \textit{ty};$

Example

```
imported function
  msgPolicy (Text : in Msg,
             Output : out MsgResult);
```

Specifications

A specification declaration is the way that Guardol code is verified. Syntax:

$$\texttt{spec}\ \textit{name} = \textit{stmt}$$

where *stmt* is expected to have at least one occurrence of

$$\texttt{check}\ e$$

where *e* is a boolean expression.

- It looks like a parameterized unit test.
- It looks like some code sprinkled with assertions.

Example: Tree Guard

Traverses and enforces a security policy over a tree of messages (strings), calling out to a platform-supplied dirty-word operation to scrub each message in the tree.

```
package MsgTree =
begin
 type Msg = string;
 ...
end
```

Tree Guard (contd.)

Declare the type of message trees.

```
type Tree =
 { Leaf
 | Node: [Value:Msg;  Left:Tree;  Right:Tree]
 };
```

Declare type encapsulating success/failure of tree guard.

```
type TreeResult =
 { OK : Tree
 | Audit : string
 };
```

Tree Guard (contd.)

Declare externally-supplied operation on messages, which succeeds (with possibly scrubbed message) or fails (with audit string).

```
type MsgResult = {Pass : Msg | Fail : string};

imported function
    msgPolicy (Text : in Msg,
               Output : out MsgResult);
```

The guard needs to apply `msgPolicy` on all messages in the tree, emitting an audit if `msgPolicy` returns `Fail`.

Tree guard on a slide

```
function Guard (Input : in Tree, Output : out TreeResult) =
 begin
     var ValueResult : MsgResult;
         LeftResult,RightResult : TreeResult;
 in
 match Input with
   Tree’Leaf => Output := TreeResult’OK(Tree’Leaf);
   Tree’Node node =>
   begin
     msgPolicy(node.Value, ValueResult);
     match ValueResult with
       MsgResult’Fail A  => Output := TreeResult’Audit(A);
       MsgResult’Pass ValueMsg =>
       begin
         Guard (node.Left, LeftResult);
         match LeftResult with
           TreeResult’Audit A => Output := LeftResult;
           TreeResult’OK LeftTree =>
           begin
             Guard (node.Right, RightResult);
             match RightResult with
               TreeResult’Audit A => Output := RightResult;
               TreeResult’OK RightTree =>
                    Output := TreeResult’OK(Tree’Node
                               [ Value:ValueMsg, Left:LeftTree, Right:RightTree ]);
 end end end end
```

Tree Guard (contd.)

The algorithm works by case analysis on how the input tree can be constructed. If `Input` is a `Leaf`, then it is OK. Otherwise, it must be a `Node`, and the code has to

- scrub the message at the node, by invoking `msgPolicy`
- analyze the left subtree;
- analyze the right subtree;
- collect up the results.

Control flow via pattern matching

ML-style pattern-matching over datatype constructors is used to analyze the structure of `Input`.

```
match Input with
  Tree'Leaf => Output := TreeResult'OK(Tree'Leaf);
  Tree'Node node =>
            ... node.Value ...
              ... node.Left   ...
                ... node.Right ...
```

In the second clause, we use the variable `node` to name the node contents. We can then use record projections to access subcomponents of `Input`.

Externals

Now we want to analyze the contents of a node. First, we call the external procedure, obtaining the verdict in `ValueResult`. If it's an audit, then turn it into a tree-level audit, and return immediately. Otherwise, the scrubbed message is named `ValueMsg` and processing continues.

```
Tree'Node node =>
begin
  msgPolicy(node.Value, ValueResult) ;
  match ValueResult with
    MsgResult'Audit A
       => Output := TreeResult'Audit(A);
    MsgResult'Ok ValueMsg
       =>  ...
```

Recursion

We recurse into left subtree. If audit happens anywhere in it, propagate the audit. Otherwise, recurse into right subtree. If audit happens, propagate. Otherwise we have scrubbed trees named `LeftTree`, and `RightTree`.

```
begin
  Guard (node.Left, LeftResult);
  match LeftResult with
     TreeResult'Audit A => Output := LeftResult;
      TreeResult'OK LeftTree =>
         begin
           Guard (node.Right, RightResult);
           match RightResult with
             TreeResult'Audit A
               => Output := RightResult;
             TreeResult'OK RightTree
               => ...
```

Return scrubbed tree

The message, left subtree, and right subtree have all been scrubbed. Time to return a scrubbed tree comprising them.

```
Output := TreeResult'OK
            (Tree'Node
                [Value : ValueMsg,
                 Left  : LeftTree,
                 Right : RightTree]);
```

That finishes the definition of the tree guard.

Tree Guard specification

Our tree guard example is quite general because it is parameterized by the dirty-word policy. The specification that we want to hold is, roughly,

> *If we run the guard successfully on a tree of messages, then every message in the result is clean, i.e., scrubbing again changes nothing.*

- This is a disguised form of *idempotence*.
- Idempotence of the guard depends on idempotence of the external dirty-word operation!

Tree Guard specification (contd.)

Experience in working with developers tells us that we don't want to use a logic language to write specifications. First, a predicate that returns true if a tree doesn't change under application of **msgPolicy**:

```
function Tree_Stable (MT : in Tree) returns Output:bool
 begin var R : MsgResult;
 in
  match MT with
   Tree'Leaf => Output := true;
   Tree'Node node =>
      msgPolicy(node.Value, R);
      match R with
        MsgResult'Pass M => Output := node.Value = M;
        MsgResult'Fail A => Output := false;

      Output := Output and Tree_Stable(node.Left)}
                       and Tree_Stable(node.Right);
 end
```

Tree Guard specification (contd.)

Then some code that **msgPolicy** is idempotent on its input string:

```
function msgPolicy_Idempotent(M : in Msg)
         returns Output : bool =
 begin var R1,R2 : MsgResult;
 in
   msgPolicy(M, R1);
   match R1 with
     MsgResult'Fail A => Output := true;
     MsgResult'Pass M2 =>
       msgPolicy(M2, R2);
       match R2 with
          MsgResult'Fail A => Output := false;
          MsgResult'Pass M3 => Output := M2 = M3;
end
```

Tree Guard specification (contd.)

Now we run the guard and check that the resulting tree is stable. Proving this goal requires that the external function is idempotent on all strings:

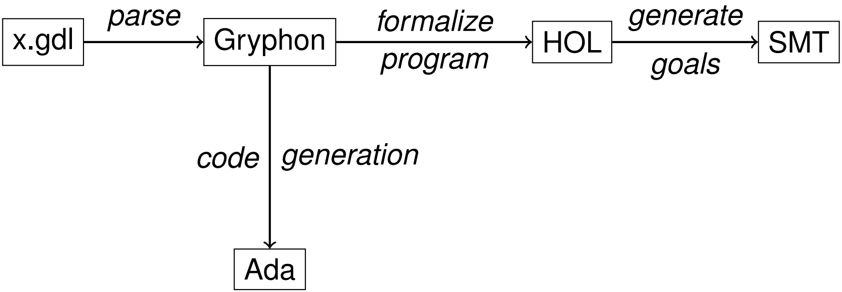
```
spec Guard_Correct =
 begin var MT : Tree;
           RT : TreeResult;
 in
   if (forall (M:Msg). msgPolicy_Idempotent M)
    then Guard(MT, RT);
         check Result_OK(RT);
    else skip;
 end

function Result_OK(TR:in TreeResult) returns Output:bool
 = begin match TR with
           TreeResult'Audit A => Output := true;
           TreeResult'OK t => Output := Tree_Stable(t);
   end
```

Part 3

The Guardol Verification System

Guardol System Diagram



The Guardol system supports code generation and verification.

Verification

If the user chooses to verify the code in `x.gdl`, the HOL4 and SMT systems become involved.

- **HOL4** is an implementation of higher order logic. It is well-suited to give semantics to programming languages.
- SMT (Satisfiability Modulo Theories) is a framework for coordinated proof using decision procedure technology.

Guardol operational semantics

The operational semantics of Guardol describes program evaluation. The semantics takes the form of an inductively defined judgement saying how statements alter the program state. The formula:

$$\mathbf{STEPS}\ \Gamma\ \mathit{prog}\ (\mathbf{Normal}\ s_1)\ (\mathbf{Normal}\ s_2)$$

says “evaluation of program *prog* beginning in state s_1 terminates and results in state s_2”.

- This is a so-called *big-step* semantics.
- Γ is an environment binding procedure names to procedure bodies.

Verification

The operational semantics of Guardol has been formalized in HOL4. The first step along the verification path is the translation of the types and code in `x.gdl` to formal analogues in this theory.

- One could reason in HOL4 about such programs, directly using the operational semantics.
- But that’s a chore.
- Instead, we use HOL as a **semantical conduit** to fully automated proof.
- We utlimately use the high automation promised by systems such as OpenSMT and CVC4

Semantic translation

- HOL mapping
  - Guardol types go to HOL types
  - Guardol expressions go to HOL terms
  - Guardol statements go to AST nodes
  - Guardol procedures go to HOL functions. Recursive procedures map to recursive functions.
- Decompilation proves equivalence between
  - a Guardol program under operational semantics and
  - a *footprint function* representing the program

Decompilation into logic

A decompilation theorem

$$
\begin{array}{l}
\vdash \forall s_1\ s_2.\ \forall x_1 \ldots x_k. \\
\quad s_1.proc.v_1 = x_1 \wedge \ldots \wedge s_1.proc.v_k = x_k \wedge \\
\quad \textbf{STEPS}\ \Gamma\ \textbf{code}\ (\textbf{Normal}\ s_1)\ (\textbf{Normal}\ s_2) \\
\quad \Rightarrow \\
\quad \texttt{let}\ (o_1, ..., o_n) = \textbf{fn}(x_1, \ldots, x_k) \\
\quad \texttt{in}\ s_2 = s_1\ \texttt{with}\{y_1 := o_1, \ldots, y_n := o_n\}
\end{array}
$$

relates evaluation of a program **code** with a footprint function **fn** which captures the behavior of the program.

Proving decompilation theorems

Decompilation theorems allow reasoning about execution to be replaced by reasoning about footprint functions.

- Automatically proved
- Bottom-up approach
- Essentially symbolic evaluation of program, using env. of decompilation theorems to summarize behavior of procedures
- Induction on recursion structure needed for recursive procedures.

Transformation Example

The correctness goal for our Tree Guard example is

```
∀u1 u2 MT RT.
   (u1.Guard_Correct.MT = MT) ∧
   (u1.Guard_Correct.RT = RT)) ∧
   STEPS Gamma code (Normal u1) (Normal u2)
  ==> u2.Guard_Correct.V
```

The goal resulting from applying the decompilation theorem:

```
  (∀M. msgPolicy_IdempotentFn ext M)
 ==>
  Result_OKFn ext (GuardFn ext v1)
```

Decision procedures for functional programs

We want to automate much or all of the reasoning about Guardol programs.

- In general that's not possible (Turing, Rice, *etc*)
- But, new decision procedures for functional programs over recursive datatypes have recently emerged and we have implemented one of them, due to Suter and Kuncak.
- Reference: Suter, Koeksal, and Kuncak. Satisfiability Modulo Recursive Programs. SAS 2011 Proceedings.

Automated reasoning about Guardol programs

The goals we are interested in are of the form

$$H_1 \wedge \ldots \wedge H_m \Rightarrow P\ (\ \mathbf{cata}\ \ (\mathbf{Guard}\ x_1 \ldots x_n))$$

where *cata* is a so-called *catamorphism* (a.k.a *fold*) and the fold maps the output of the Guard into a decidable theory.

- Under certain technical conditions, this class of formulas is decidable by SK.
- BUT FIRST we need to
    - induct with scheme for **Guard**,
    - expand **Guard** once, and
    - instantiate any quantifiers in the hypotheses

Catamorphism aka Folds

A catamorphism is a simple pattern of recursion in which an operator

$$\textbf{op} : (\alpha \to \beta \to \beta) \to \alpha\ \textbf{list} \to \beta \to \beta$$

is used to crunch a recursive branching structure down into a single value.

$$\begin{aligned}
\textbf{fold}\ (+)\ [x_1, \ldots, x_n]\ 0 &= x_1 + \cdots + x_n + 0\\
\textbf{fold}\ (*)\ [x_1, \ldots, x_n]\ 1 &= x_1 * \cdots * x_n * 1\\
\textbf{fold}\ (\mathit{cons})\ [x_1, \ldots, x_n]\ [] &= [x_1, \ldots, x_n]
\end{aligned}$$

Note also that a *sorted* predicate on lists is a catamorphism.

Catamorphism

Tree_Stable is a catamorphism on **Tree**.

```
function Tree_Stable (MT : in Tree) returns Output:bool
 begin var R : MsgResult;
 in
  match MT with
   Tree'Leaf => Output := true;
   Tree'Node node =>
      msgPolicy(node.Value, R);
      match R with
        MsgResult'Pass M => Output := node.Value = M;
        MsgResult'Fail A => Output := false;

      Output := Output and Tree_Stable(node.Left)
                       and Tree_Stable(node.Right);
 end
```

Implementing SK

We (Whalen and Pham) have been implementing the SK decision procedure

- Integrating the decision procedure into public Satisfiability Modulo Theories (SMT) frameworks
- Not a smooth fit because the SK procedure needs to work on terms before the purification step.
- Extended the d.p. to handle mutual recursion
- Z3, OpenSMT, CVC4 used so far.

Remark on guard properties

Guardol programs get translated to mathematical functions in our verification path. This yields good news and bad news.

- Good news: we can apply decision procedures to prove specifications.
- Bad news: the properties we can show are **extensional**, but some properties of interest are **intensional**
- On the bright side: we know how to deal with the bad news. ("Model checking information flow", Whalen and Greve).

Extensional vs. Intensional

- Extensional: only the input/output of a computation is visible (what not how).
- Intensional: how the computation works is visible.

Example

Extensional. The procedure sorts a list of integers.

Intensional. The procedure sorts a list of integers in-place.

Example

Extensional. Scrubbing a message twice gives the same result as scrubbing it once.

Intensional. The guard analyzes every component of every message.

Summary

The Guardol system is a domain-specific language aimed at advancing the state of the art in developing and proving correctness of high-assurance guards.

- Ada code for a number of guards can be generated.
- Rules for specific guards can be generated.
- Specifications of Guardol programs are soundly and automatically translated into goals for automatic proof.
- A reasonable class of guard properties (but not all) can be be decided by our implementation.

THE END