Formal Methods in Differential and Linear Trail Search

Benoît Viguier
October 7, 2016
Overview

Introduction

Keccak

Differential Cryptanalysis

Semantics of Trees and Iterators

Proven Iterator

Conclusion
Introduction
Hashing vs Encryption

Hashing

Plaintext -> Hash

Encryption

Ciphertext -> Hash
Second pre-image attack

Hashing

I will pay 5 $.

0xAB8924

I will pay 5000 $.

0xAB8924

Collision

attack
Keccak
Sponge construction + invertible permutation $f$ named $\text{KECCAK-f}[b]$.

Figure 1: A sponge construction

bit rate ($r$) + capacity ($c$) = width ($b$)

$\text{KECCAK-f}[1600] = (\iota \circ \chi \circ \pi \circ \rho \circ \theta)^{24}$ and $b = 1600$
Figure 2: Keccak[200] state
Keccak-\textit{f}: \( \theta \)

Linear mixing layer on column parity.

\[
p[x, z] : \bigoplus_{y=0}^{4} a[x, y, z]
\]

\[
E(x, z) = \mathcal{P}[x-1, z] \oplus \mathcal{P}[x+1, z-1]
\]

\textbf{Figure 3:} Application of \( \theta \) to a state.

\( \mathcal{P} \) is called the parity plane.
Keccak-\( f \): \( \rho \) and \( \pi \)

Bit-wise cyclic shift rotation on lanes.

Figure 4: The \( \rho \) transformation

Lane transposition.

Figure 5: The \( \pi \) transposition
Keccak-\( f \): \( \chi \)

Non-linear mapping \( f \) algebraic degree of 2 which operates on rows.

**Figure 6:** The \( \chi \) transformation
Differential Cryptanalysis
Given an input difference $\Delta_1$, chances are that a difference $\Delta_2$ will occur. It can be associated with a probability: $P[(\Delta_1 \Rightarrow \Delta_2)]$. 

**Figure 7:** A differential $(\Delta_1 \Rightarrow f \Delta_2)$
Figure 8: A trail \((\Delta_0 \Rightarrow f \Rightarrow \Delta_1 \Rightarrow f \Rightarrow \Delta_2)\)

**Goal:** Find a trail \((\Delta_0 \Rightarrow f \Rightarrow \cdots \Rightarrow \Delta_2)\) such as \(\Delta_n = 0\) \((\Leftrightarrow \text{collision})\).
There are $2^{1600} - 1$ input differences possible for Keccak-$f[1600]$.

Estimated number of hydrogen atoms in the Universe: $\approx 2^{265}$
Figure 9: Tree decomposition of the search.
What are the verifications needed?

- orbitals (involution, order)
- columns assignement (order)
- runs (order, z-canonicity factorization)
- and more...

What are the difficulties?

- C++ $\Rightarrow$ no VST, no FRAMA-C, no Why3.
- Huge source code!

What have been done during this Internship?

- Using Hoare Logic.
- Orbitals: involution and order

The time I would have spent on more proofs would not have been compensated by the gain of the correction.
Figure 10: Tree decomposition of the search.
Figure 10: Tree decomposition of the search.
Figure 10: Tree decomposition of the search.
Semantics of Trees and Iterators
Section trees.
  Variable (X : Type).

  (* we do not want the too weak Coq generated induction principles *)
Unset Elimination Schemes.

Inductive Tree : Type :=
  node : X → list Tree → Tree.
Set Elimination Schemes.

Section Tree_ind.
  Variable P : Tree → Prop.
  Hypothesis HP : ∀ a ll,
    (∀ x, In x ll → P x) → P (node a ll).
  Definition Tree_ind : ∀ t, P t.
End Tree_ind.
End trees.

Code 1: Tree definition

Induction principle:

1. prove the property for a tree with no children.

2. Assume that the property is True for all children, prove it for the parent.
**Definition** Path := list nat.

**Definition** getNode (p:Path) (t:Tree X) : option (Tree X) := ... 

---

**Code 2:** Path definition

Each node from the tree can be accessed by a path specified as the list of the index of the child to consider.

- \([\ ]\) returns \textit{root}.
- \([0]\) returns \textit{N}_1.
- \([0,0]\) returns \textit{N}_2.
- \([1,0]\) returns \textit{N}_3.

\texttt{getNode (p)} returns \texttt{Some \((n,l)\)} if a node \textit{n} with childrens \textit{l} exists or \texttt{None}.

---

**Figure 12:** Tree
Tree traversal: Moves

Figure 13: Iteration through a tree

Inductive MoveSS : Type := TO_PARENT | TO_CHILD | TO_SIBLING | VISITED.

Code 3: Definition of the movements
We can use Small-step semantics to specify rules over moves.

\[ it : (\text{move}, \text{path}, \text{visited nodes}) \rightarrow (\text{move'}, \text{path'}, \text{visited nodes'}) \]

**Inductive** iterator\_smallstep\_v X:

\[ \text{Tree } X \rightarrow \text{MoveSS } \ast \text{Path } \ast (\text{Visited } X) \rightarrow \text{MoveSS } \ast \text{Path } \ast (\text{Visited } X) \rightarrow \text{Prop} \]

\[\begin{align*}
& \text{(visit\_up)} \\
& \quad \frac{\text{(TO\_PARENT, } p, v) \rightarrow (\text{VISITED, } p, v)}{}
\end{align*}\]

\[\begin{align*}
& \text{(visit\_no\_sons)} \\
& \quad \frac{m \neq \text{TO\_PARENT} \quad m \neq \text{VISITED} \quad \text{getNode}(p) \mapsto \text{Some } (n, [])}{(m, p, v) \rightarrow (\text{VISITED, } p, n :: v)}
\end{align*}\]

\[\begin{align*}
& \text{(down)} \\
& \quad \frac{m \neq \text{TO\_PARENT} \quad m \neq \text{VISITED} \quad \text{getNode}(p) \mapsto \text{Some } (n, l) \quad l \neq []}{(m, p, v) \rightarrow (\text{TO\_CHILD, } 0 :: p, n :: v)}
\end{align*}\]

\[\begin{align*}
& \text{(up)} \\
& \quad \frac{\text{getNode}(h :: p) \mapsto \text{Some } (n, l) \quad \text{getNode}(h + 1 :: p) \mapsto \text{None}}{(\text{VISITED, } h :: p, v) \rightarrow (\text{TO\_PARENT, } p, v)}
\end{align*}\]

\[\begin{align*}
& \text{(next)} \\
& \quad \frac{\text{getNode}(h + 1 :: p) \mapsto \text{Some } (n, l)}{(\text{VISITED, } h :: p, v) \rightarrow (\text{TO\_SIBLING, } h + 1 :: p, v)}
\end{align*}\]
Tree traversal: Rules

1. **visit_up**
   If we just went back to the parent, the next move is VISITED.

2. **visit_no_sons**
   If the node does not have children, the next move is VISITED.

3. **down**
   If the node has a child (and the node is not VISITED), the next move is TO_CHILD.

4. **up**
   If the node is VISITED and has no siblings, the next move is TO_PARENT.

5. **next**
   If the node is VISITED and has siblings, the next move is TO_SIBLING.

---

**Figure 14:** Iteration rules applied to tree traversal
Tree traversal: Theorems

Iterator is deterministic:

\[ \forall \text{move path visited}, \]
\[ (\forall \text{move}_1 \text{ path}_1 \text{ visited}_1, \text{it} : (\text{move}, \text{path}, \text{visited}) \rightarrow (\text{move}_1, \text{path}_1, \text{visited}_1) \land \]
\[ \forall \text{move}_2 \text{ path}_2 \text{ visited}_2, \text{it} : (\text{move}, \text{path}, \text{visited}) \rightarrow (\text{move}_2, \text{path}_2, \text{visited}_2)) \Rightarrow \]
\[ \text{move}_1 = \text{move}_2 \land \text{path}_1 = \text{path}_2 \land \text{visited}_1 = \text{visited}_2 \]

Iterator’s traversal is complete:

\[ \text{it} : (\text{TO\_CHILD}, [], []) \rightarrow^\ast (\text{VISITED}, [], \text{visited}) \]

where \text{visited} is the list of the values of all the nodes.
Tree pruning

**Figure 15:** Tree pruning.
Figure 15: Tree pruning.
Tree traversal: Rules Augmented

The iterator should also cut branches of the tree when some conditions are met (simulated by the evaluation of a function $B : node \to \mathit{Bool}$)

\[
\begin{align*}
\text{(visit\_up)} & \quad (\text{TO\_PARENT}, p, v) \to (\text{VISITED}, p, v) \\
\end{align*}
\]

\[
\begin{align*}
\text{(visit\_no\_sons\_true)} & \quad \begin{array}{c}
m \neq \text{TO\_PARENT} \\
m \neq \text{VISITED} \\
\end{array} \quad \begin{array}{c}
\text{getNode } p \mapsto \text{Some} (n, []) \\
B \ n = \text{True} \\
\end{array} \quad (m, p, v) \to (\text{VISITED}, p, n :: v) \\
\end{align*}
\]

\[
\begin{align*}
\text{(down)} & \quad \begin{array}{c}
m \neq \text{TO\_PARENT} \\
m \neq \text{VISITED} \\
\end{array} \quad \begin{array}{c}
\text{getNode } p \mapsto \text{Some} (n, l) \\
l \neq [] \\
B \ n = \text{True} \\
\end{array} \quad (m, p, v) \to (\text{TO\_CHILD}, 0 :: p, n :: v) \\
\end{align*}
\]

\[
\begin{align*}
\text{(down\_forbidden)} & \quad \begin{array}{c}
m \neq \text{VISITED} \\
\end{array} \quad \begin{array}{c}
\text{getNode } p \mapsto \text{Some} (n, l) \\
B \ n = \text{False} \\
\end{array} \quad (m, p, v) \to (\text{VISITED}, p, v) \\
\end{align*}
\]

\[
\begin{align*}
\text{(up)} & \quad \begin{array}{c}
\text{getNode } (h :: p) \mapsto \text{Some} (n, l) \\
\text{getNode } (h + 1 :: p) \mapsto \text{None} \\
\end{array} \quad (\text{VISITED}, h :: p, v) \to (\text{TO\_PARENT}, p, v) \\
\end{align*}
\]

\[
\begin{align*}
\text{(next)} & \quad \begin{array}{c}
\text{getNode } (h + 1 :: p) \mapsto \text{Some} (n, l) \\
\end{array} \quad (\text{VISITED}, h :: p, v) \to (\text{TO\_SIBLING}, h + 1 :: p, v) \\
\end{align*}
\]
Proven Iterator
Toward an Iterator

The iterator (*manager*) should provide the next move with the minimum of required information.

- Path
- is the last move *toward the parent*?
- move VISITED will be skipped.
Iterator in Gallina

(*
what are the assumptions before going in this function? Make no such assumption.
Only need to know only one thing: was the last move TO_PARENT (last_up = true)?
*)

Definition manager X (t:Tree X) (B:X → bool) (pl:option (Path*bool)) :
option (MoveSS) :=
match pl with
| None ⇒ None
| Some (p,last_up) ⇒ match getNode p t with
  | None ⇒ None
  | _ ⇒
    if andb (NodeValid p t B) (negb last_up) then (\* \∧ \*)
      if ChildExists p t then (\* | \*)
        Some TO_CHILD (\* | \*)
        else (\* | \*)
          if SiblingExists p t then (\* | \* This part will be \*)
            Some TO_SIBLING (\* | \* directly translated \*)
          else (\* | \*)
            Some TO_PARENT (\* | \*)
            else (\* | \*)
              if SiblingExists p t then (\* | \*)
                Some TO_SIBLING (\* | \*)
              else (\* | \*)
                Some TO_PARENT (\* \∨ \*)
    end
end.

Code 4: Given a path we can select the next move
Move Manager::next_move() {
    if (path->isNodeValid() && !is_last_move_to_parent) {
        if (path->hasChild()) {
            return TO_CHILD;
        } else {
            if (path->hasSiblings()) {
                return TO_SIBLING;
            } else {
                return TO_PARENT;
            }
        }
    } else {
        if (path->hasSiblings()) {
            return TO_SIBLING;
        } else {
            return TO_PARENT;
        }
    }
}

Code 5: Definition of the Manager in C++
The C++ Gallina Equivalence

**Figure 16: Code Gallina**

```gallina
Definition manager X (t:Tree X) (B:X -> bool) =
  option (MoveSS) :=
  match pl with
  | None => None
  | Some (p,last_up) => match getNode p t w
  | None => None
  | _ =>
    if andb (NodeValid p t B) (negb last_u)
    if ChildExists p t then
      Some TO_CHILD
    else
      if SiblingExists p t then
        Some TO_SIBLING
      else
        Some TO_PARENT
    else
      if SiblingExists p t then
        Some TO_SIBLING
      else
        Some TO_PARENT
  end
end.
```

**Figure 17: Code C++**

```cpp
Move Manager::next_move() {
  if (path->isNodeValid() && !is_
    if (path->hasChild()) {
      return TO_CHILD;
    }
  else {
    if (path->hasSiblings()) {
      return TO_SIBLING;
    }
  return TO_PARENT;
  }
  }
  }
  else {
    if (path->hasSiblings()) {
      return TO_SIBLING;
    }
  return TO_PARENT;
  }
  }
```
Equivalence

∀ tree path last_up move path',

\[ \text{manager}(\text{tree}, \text{path}, \text{last_up}) \leftrightarrow \text{move} \land \text{apply}(\text{move}, \text{path}) \leftrightarrow \text{path'} \Rightarrow \]

\[ \text{it} : (\ldots, \text{path}, \ldots) \rightarrow (\text{move}, \text{path'}, \ldots) \]

**Theorem** managerEqSemantic :

\[ \forall X (B:X \rightarrow \text{bool}) \ (\text{tree}:\text{Tree} \ X) \ (m \ m':\text{MoveSS}) \ (p \ p':\text{Path}) \ \text{last_up} \ \text{last_up}', \]

(* Define the equivalence between the last movement and the last_up boolean value as hypotheses. *)

\[
\begin{align*}
\text{last_up'} = \text{true} \leftrightarrow (m' = \text{TO_PARENT}) \\
\text{last_up} = \text{false} \leftrightarrow (m = \text{TO_CHILD} \lor m = \text{TO_SIBLING}) \\
\text{last_up} = \text{true} \leftrightarrow (m = \text{TO_PARENT}) \land \text{NodeExists (0::p) tree = true}
\end{align*}
\]

(* Apply the move to the path and return the boolean value to for the manager *)

\[ \text{applyMove} \ p \ m' = \text{Some} \ (p', \text{last_up'}) \rightarrow \]

(* manager hypothesis *)

\[ \text{manager} \ \text{tree} \ B \ (\text{Some} \ (p, \text{last_up})) = \text{Some} \ m' \]

→

(* Either we have an intermediate VISITED step *)

\[ \text{iterator_nv} \ B \ \text{tree} \ (m, p) \ (\text{VISITED}, p) \land \text{iterator_nv} \ B \ \text{tree} \ (\text{VISITED}, p) \ (m', p') \]

(* Or we are right *)

\[ \lor \text{iterator_nv} \ B \ \text{tree} \ (m, p) \ (m', p'). \]

**Code 6:** Theorem of the implication between the manager and the semantic iterator
Figure 18: Tree decomposition of the search.
**Figure 19:** Tree/Path and Stack equivalence. The head of the list/stack is in red.
What do we have to trust?
What do we trust?

- Calculus of Inductive Construction
- Specification and Small-step semantics
- Tree implementation and specification: **WEAK LINK**
- Translation from **GALLINA** to C++
- GCC
- Coq kernel, Ocaml compiler, Ocaml Runtime, CPU.
Conclusion
Conclusion

From Code to Proofs:

The orbitals iterator has been proven correct (involution + order) with Hoare logic.

From Proofs to Code:

- Specification of generic tree
- Specification of an iterator in Small-step semantics
- Definition of an abstract iterator (manager) which fully traverse any given tree.

By providing such iterator, we reduce the trust to the tree definition/construction.
Questions?
Thank you!
We define the weight \((w)\) of a differential as follow.

\[
P[(\Delta_1 \Rightarrow \Delta_2)] = \frac{1}{2^w}
\]

The weight of a trail \(Q = (\Delta_0 \Rightarrow \cdots \Rightarrow \Delta_n)\) is the sum of the weight of its differentials.

\[
w(Q) = \sum_{i=0}^{n-1} w(\Delta_i \Rightarrow \Delta_{i+1})
\]

**Remark:** Affine applications have no influence on the probabilities of differentials.
Let $Q$ be a trail of differences $a_0, a_1, \ldots a_n$:

$$Q = a_0 \xrightarrow{\chi \circ \pi \circ \rho \circ \theta} a_1 \xrightarrow{\chi \circ \pi \circ \rho \circ \theta} \ldots \xrightarrow{\chi \circ \pi \circ \rho \circ \theta} a_n$$

$$Q = a_0 \xrightarrow{\pi \circ \rho \circ \theta} b_0 \xrightarrow{\chi} a_1 \xrightarrow{\pi \circ \rho \circ \theta} \ldots \xrightarrow{\chi} a_n$$

Because $\pi \circ \rho \circ \theta$ is linear, we have $w(a_i \xrightarrow{\pi \circ \rho \circ \theta} b_i) = 0$. Therefore:

$$Q = \sum_{i=0}^{n-1} w(b_i \xrightarrow{\chi} a_{i+1})$$

**The weight depend only on the propagation of $b_i$ through $\chi$.**
Affine applications have no influence on the probabilities of differentials.

Proof:

• \( K \in \text{GF}(2)^n \) a constant;
• \( A \) a permutation matrix of \( \text{GF}(2)^n \);
• \( f : \text{GF}(2)^n \rightarrow \text{GF}(2)^n \) such as \( f(x) = Ax + K \);
• \( \Delta \) and \( \Delta' \) two differences

\[
P(\Delta \xrightarrow{f} \Delta') > 0 \iff \exists t, \Delta' = f(t) + f(t + \Delta)
\]
\[
\iff \exists t, \Delta' = At + K + A(t + \Delta) + K
\]
\[
\iff \Delta' = A\Delta
\]

Therefore the probability of a differential over an affine application is 1. \( \square \)
**Table 1:** Space of possible output differences, weight, and Hamming weight of all row differences.

<table>
<thead>
<tr>
<th>input difference</th>
<th>offset</th>
<th>propagation through $\chi$ base elements</th>
<th>$w(.)$</th>
<th>$| . |$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00000</td>
<td>00010 00100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00001</td>
<td>00001</td>
<td>00010 00100</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>00011</td>
<td>00001</td>
<td>00010 00100</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>00101</td>
<td>00001</td>
<td>00010 01100 10000</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10101</td>
<td>00001</td>
<td>00010 01100 10001</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>00111</td>
<td>00011</td>
<td>00010 00100</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>01111</td>
<td>00011</td>
<td>00010 00100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>11111</td>
<td>00001</td>
<td>00011 00110 01100 11000</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

**Adding active bits to the state will never decrease the weight.**
Orbitals

\[ \rho[x, z] = \bigoplus_{y=0}^{4} a[x, y, z] \]

 parity plane \( \mathcal{P} \)

**Figure 21:** State and Parity.

**Figure 22:** Orbitals, active bits are coloured.
Small-step semantics and big-step semantics

**Small steps**
Rules which specify from configuration $c$ and state $s$, one can go to configuration $c'$ and state $s'$.

$$< c, s > \rightarrow < c', s' >$$

$$< c, s > \rightarrow^* < \delta, \sigma >$$

**Big steps**
Rules which specify the entire transition from a configuration $c$ and state $s'$ to a final state $\sigma$.

$$< c, s > \Downarrow \sigma$$

**Equivalence big steps - small steps**

$$< c, s > \rightarrow^* < \delta, \sigma > \iff < c, s > \Downarrow \sigma$$
Tree traversal: Proof (1/6)

\[ it : (\text{TO\_CHILD}, [], []) \rightarrow^* (\text{VISITED}, [], \text{visited}) \]

Provide genericity:

\[ \forall \text{path pred}, it : (\text{TO\_CHILD}, \text{path}, \text{pred}) \rightarrow^* (\text{VISITED}, \text{path}, \text{visited} :: \text{pred}) \]

By induction:

\[ \ldots \]

**Figure 23:** \( \text{getNode path} = \text{Some}(n, []) \)

\[ n \]

\[ \ldots \]

\[ n_n \ldots n_2 n_1 \]

**Figure 24:** \( \text{getNode path} = \text{Some}(n, l) \)
Goal:

\[ \forall path \ pred, it : (\text{TO\_CHILD}, path, pred) \rightarrow^* (\text{VISITED}, path, visited :: pred) \]

By induction:

\( getNode \ path = \text{Some}(n, []) \) ✓

\( getNode \ path = \text{Some}(n, l) \)

![Diagram](image)

**Figure 25:** \( getNode \ path = \text{Some}(n, []) \)
Goal:

\[ \forall \text{ path pred, it} : (\text{TO\_CHILD, path, pred}) \rightarrow^* (\text{VISITED, path, visited :: pred}) \]

By induction:

\[ \text{getNode path} = \text{Some}(n, []) \checkmark \]

\[ \text{getNode path} = \text{Some}(n, l) \]

\[ \text{Figure 26: getNode path} = \text{Some}(n, l) \]
Goal:

\[ \forall \text{path pred, it : (TO\_CHILD, path, pred)} \rightarrow^* (\text{VISITED, path, visited :: pred}) \]

By induction:

\[ \text{getNode path} = \text{Some}(n, []) \checkmark \]
\[ \text{getNode path} = \text{Some}(n, l) \]

Figure 27: \( \text{getNode path} = \text{Some}(n, l) \)
Goal:

\[ \forall \text{path pred, it : (TO\_CHILD, path, pred)} \rightarrow^* (\text{VISITED, path, visited :: pred}) \]

By induction:

\[ \text{getNode \ path} = \text{Some}(n, []) \checkmark \]

\[ \text{getNode \ path} = \text{Some}(n, l) \]

Figure 28: getNode \ path = Some(n, l)
Goal:

\[ \forall \text{path pred, it : (TO\_CHILD, path, pred)} \rightarrow^* (\text{VISITED, path, visited :: pred}) \]

By induction:

- getNode path = Some(n, []) ✓
- getNode path = Some(n, l) ✓

\[ (\text{TO\_CHILD, pred}) \rightarrow^* (\text{VISITED, V :: \ldots :: V :: V :: n :: pred}) \]

Figure 29: getNode path = Some(n, l)