Exploration of Reflective ASMs for Genetic Algorithms and Security

Linjie Tong[1], Ke Xu[1], Jiarun Hu[1], Flavio Ferrarotti[2], and Klaus-Dieter Schewe[1]

[1] Zhejiang University, UIUC Institute, China,
[linjie.19|ke.19|jiarun.19]@intl.zju.edu.cn, kdschewe@acm.org
[2] Software Competence Centre Hagenberg, Austria, flavio.ferrarotti@scch.at

Abstract. Recently, a behavioural theory for reflective algorithms has been developed, which shows that reflective (sequential) ASMs (rASMs) capture all reflective (sequential) algorithms. In this paper we explore the expressive power of rASMs for genetic algorithms and security. We first show that all genetic algorithms are captured by rASMs. Then we elaborate recombinative simulated annealing as a specific example of a genetic algorithm specified an rASM. We further show how rASMs can support hardware-software binding, which can be used for copy protection. We exploit the logic of rASMs to express desirable properties for this application.

Keywords: reflective Abstract State Machines, genetic algorithms, recombinative simulated annealing, hardware-software binding

1 Introduction

The concept of *linguistic reflection* in programming refers to the ability of a program to change its own behaviour. It is as old as any higher programming language; it appeared already in the 1950s in LISP. While it is difficult to maintain control of the desired behaviour of a program when this behaviour is subject to on-the-fly changes, controlled versions of reflection have shown to be extremely valuable for persistent programming (see e.g. the work of Stemple et al. [14]).

In a recent article [10] the last two authors developed a behavioural theory of reflective algorithms, first formulated and proven for the case of sequential algorithms. The theory shows that all reflective sequential algorithms are captured by reflective sequential Abstract State Machines (ASMs), so it becomes possible to specify the behaviour of reflective programs in a rigorous and controllable way. Furthermore, an associated logic for reflective ASMs (rASMs) was developed in [11] (not restricted to the sequential case) by extending the logic of non-deterministic ASMs. By means of this logic desirable properties of the dynamic behaviour of rASMs can be formalised statically and verified. These theories provide an important contribution to making adaptive systems reliable.

As a generic example genetic algorithms [4] were used in [11]. Genetic algorithms are the most prominent representatives of natural computing, an area where algorithms are designed in a way to mimick processes in nature. A common motivation is the capture of complex adaptive systems [5]. This area developed quite independently from the work above on linguistic reflection.

In this article we further explore the expressive power of rASMs. We demonstrate in Section 3 that all genetic algorithms are captured by rASMs. Then in Section 4 we elaborate recombinative simulated annealing [7] as a specific example of a genetic algorithm specified by an rASM, which provides a refinement. We further show in Section 5 how rASMs can be used to specify hardware-software binding, by means of which security, in particular copy protection, can be supported. For this application we exploit the logic of rASMs to precisely define desirable properties. The article is complemented by a brief introduction of general rASMs in Section 2 and concluding remarks in Section 6.

2 Reflective ASMs

We assume general familiarity with ASMs as defined in [3]. The extension to reflective ASMs requires to define a background structure that covers trees and operations on them, a dedicated variable *self* that takes as its value a tree representation of an ASM signature and rule, and the extension of rules by partial updates. Details have been presented in [11], but here we also make use of non-deterministic choice rules. We also exploit partial updates in ASMs [13].

Let Σ be an ASM signature, i.e. a set of function symbols. Partial assignments are defined as follows: Whenever $f \in \Sigma$ has arity n and *op* is an operator of arity $m+1$, t_i $(i = 1, \dots, n)$ and t'_i $(i = 1, \dots, m)$ are terms over Σ, then

$$f(t_1, \dots, t_n) \leftleftharpoons^{op} t'_1, \dots, t'_m$$

is a *partial update* rule. The informal meaning is that we evaluate the terms as well as $f(t_1, \dots, t_n)$ in the current state S, then apply *op* to $\text{val}_S(f(t_1, \dots, t_n))$, $\text{val}_S(t'_1), \dots, \text{val}_S(t'_m)$. The resulting value v gives rise to an update to the location $(f, (\text{val}_S(t_1), \dots, \text{val}_S(t_n)))$, however, there may be several such updates. In general, a multiset of such updates is collapsed into an ordinary update set if possible, then the updates in the resulting update set are applied. Conditions for compatibility and the collapse of an update multiset into an update set have been elaborated in detail in [13].

Tree Structures and Algebra. A *tree* t over the set of labels L with values in the universe U comprises an unranked tree structure $\gamma_t = (\mathcal{O}_t, \prec_c, \prec_s)$ with set of nodes $\mathcal{O}_t$, child relation $\prec_c$, and sibling relation $\prec_s$, a total *label function* $\omega_t : \mathcal{O}_t \to L$, and a partial *value function* $\upsilon_t : \mathcal{O}_t \to U$ that is defined on the leaves in γ_t. Note that when we write $o_1 \prec_c o_2$, then o_2 is a child of o_1, and when we write $o_1 \prec_s o_2$, then o_1 is a left sibling of o_2.

Let T_L denote the set of all trees with labels in L, and let $root(t)$ denote the root node of a tree t. A sequence $t_1, ..., t_k$ of trees is called a *hedge*. Let ϵ denote the empty hedge, and let H_L denote the set of all hedges with labels in L. The *largest subtree* of t at node o is denoted as $\widehat{o}$. The *set of contexts* C_L over L is the set $T_{L\cup\{\xi\}}$ of trees with labels in $L \cup \{\xi\}$ ($\xi \notin L$) such that for each tree $t \in C_L$ exactly one leaf node is labelled with ξ and the value assigned to this leaf is *undef*.

The context with a single node labelled ξ is called trivial and is simply denoted as ξ. Contexts allow us to define substitution operations that replace a subtree of a tree or context by a new tree or context. This leads to *tree-by-tree substitution* $subst_{tt}(t_1, o, t_2) = t_1[\widehat{o} \mapsto t_2]$, *tree-by-context substitution* $subst_{tc}(t_1, o, \xi) = t_1[\widehat{o} \mapsto \xi]$, *context-by-context substitution* $subst_{cc}(c_1, c_2) = c_1[\xi \mapsto c_2]$, and *context-by-tree substitution* $subst_{ct}(c_1, t_2) = c_1[\xi \mapsto t_2]$.

To provide manipulation operations over trees at a level higher than individual nodes and edges, we need constructs to select arbitrary tree portions. For this we provide two selector constructs, which result in subtrees and contexts, respectively: $context : \mathcal{O}_t \times \mathcal{O}_t \to C_L$ is a partial function on pairs (o_1, o_2) of nodes with $o_1 \prec_c^+ o_2$ and $context(o_1, o_2) = subst_{tc}(\widehat{o}_1, o_2, \xi) = \widehat{o}_1[\widehat{o}_2 \mapsto \xi]$, where $\prec_c^+$ denotes the transitive closure of $\prec_c$, and $subtree : \mathcal{O}_t \to T_L$ is a function defined by $subtree(o) = \widehat{o}$.

Then the *set* $\mathbb{T}$ *of terms* over $L \cup \{\epsilon, \xi\}$ comprises label terms, hedge terms, and context terms, and the operators of the tree algebra are defined as follows:

label_hedge. The operator *label_hedge* turns a hedge into a tree with a new added root, i.e. $label\_hedge(a, t_1 \dots t_n) = a\langle t_1, \dots, t_n\rangle$.

label_context. Similarly, the operator *label_context* turns a context into a context with a new added root, i.e. $label\_context(a, c) = a\langle c\rangle$.

left_extend. *left_extend* integrates the trees in a hedge into a context extending it on the left, i.e.
$left\_extend(t_1 \dots t_n, a\langle t'_1, \dots, t'_m\rangle) = a\langle t_1, \dots, t_n, t'_1, \dots, t'_m\rangle$.

right_extend. The operator *right_extend* is defined analogously.

concat. *concat* simply concatenates two hedges, i.e.
$concat(t_1 \dots t_n, t'_1 \dots t'_m) = t_1 \dots t_n t'_1 \dots t'_m$.

inject_hedge. *inject_hedge* turns a context into a tree by substituting a hedge for ξ, i.e. $inject\_hedge(c, t_1 \dots t_n) = c[\xi \mapsto t_1 \dots t_n]$.

inject_context. The operator *inject_context* substitutes a context for ξ, i.e. $inject\_context(c_1, c_2) = c_1[\xi \mapsto c_2]$.

Reflective ASMs. For the dedicated location storing the self-representation of an ASM it is sufficient to use a single function symbol *self* of arity 0. Then in every state S the value $\mathrm{val}_S(\mathit{self})$ is a tree comprising two subtrees for the representation of the signature and the rule, respectively. That is, in the tree structure we have a root node o labelled by `self` with exactly two successor nodes, say o_0 and o_1, labelled by `signature` and `rule`, respectively. So we have $o \prec_c o_0$, $o_0 \prec_s o_1$ and $o \prec_c o_1$, where $\prec_c$ and $\prec_s$ denote, respectively, the child and sibling relationships. The subtree rooted at o_0 has as many children $o_{00}, \dots, o_{0k}$ as there are function symbols in the signature, each labelled by `func`. Each of the subtrees rooted at o_{0i} takes the form $\texttt{func}\langle\texttt{name}\langle f\rangle\texttt{arity}\langle n\rangle\rangle$ with a function name f and a natural number n. The subtree rooted at o_1 represents the rule of a sequential ASM as a tree.

The inductive definition of trees representing rules is rather straightforward. For instance, an assignment rule $f(t_1, \dots, t_n) := t_0$ is represented by a tree of the form $\mathtt{update}\langle\mathtt{func}\langle f\rangle\mathtt{term}\langle t_1 \dots t_n\rangle\mathtt{term}\langle t_0\rangle\rangle$, and a partial assignment rule $f(t_1, \dots, t_n) \Leftarrow^{op} t'_1, \dots, t'_m$ is represented by a tree of the form $\mathtt{partial}\langle\mathtt{func}\langle f\rangle\mathtt{func}\langle op\rangle\mathtt{term}\langle t_1 \dots t_n\rangle\mathtt{term}\langle t'_1 \dots t'_m\rangle\rangle$.

The *background of an rASM* is defined by a background class $\mathcal{K}$ over a background signature V_K. It must contain an infinite set *reserve* of reserve values and an infinite set Σ_{res} of reserve function symbols, the equality predicate, the undefinedness value *undef*, and a set L of labels `self`, `signature`, `rule`, `func`, `name`, `arity`, `update`, `term`, `if`, `bool`, `forall`, `var`, `par`, `choose`, `seq`, `let`, `location`, `operator`, `partial`. The background class must further define truth values and their connectives, tuples and projection operations on them, natural numbers and operations on them, trees in T_L and tree operations, and the partial function $\mathbf{I}$, where $\mathbf{I}x.\varphi$ denotes the unique x satisfying condition φ.

If B is a base set, then an *extended base set* is the smallest set B_{ext} containing B that is closed under adding function symbols from the reserve Σ_{res}, natural numbers, the terms $\mathbb{T}$ with respect to B and Σ_{res}, and terms defined by the operations of the tree algebra over Σ_{res} with labels in L as defined above. Analogously, $\mathbb{T}_{ext}$ denotes the set of terms over the signature Σ_{ext}, and then we use $\hat{\mathbb{T}}_{ext}$ to denote the union of the $\mathbb{T}_{ext}$ and the set of terms representing the rules.

Background. The background must further provide functions: $drop : \hat{\mathbb{T}}_{ext} \to B_{ext}$ and $raise : B_{ext} \to \hat{\mathbb{T}}_{ext}$ for each base set B and extended base set B_{ext}, and a derived *extraction function* $\beta : \mathbb{T}_{ext} \to \bigcup_{n\in\mathbb{N}} \mathbb{T}^n$, which assigns to each term defined over the extended signature Σ_{ext} and the extended base set B_{ext} a tuple of terms in $\mathbb{T}$ defined over Σ and B. The function *drop* turns a term in $\hat{\mathbb{T}}_{ext}$ into a value in B_{ext} making it possible to change this term by means of some update. The function *raise* does the opposite; it turns a value, e.g. a tree representing a term or a rule into a term in $\hat{\mathbb{T}}_{ext}$, so it can be executed or evaluated to yield an update. These are the standard function associated with linguistic reflection as elaborated in [14]. The function β is used to show which terms in $\mathbb{T}$ appear inside a term in $\mathbb{T}_{ext}$. This is needed to express *strong coincidence* for bounded exploration [10].

A *reflective ASM* (rASM) $\mathcal{M}$ comprises an (initial) signature Σ containing a 0-ary function symbol *self*, a background as defined above, and a set $\mathcal{I}$ of initial states over Σ closed under isomorphisms such that any two states $I_1, I_2 \in \mathcal{I}$ coincide on *self*. Furthermore, $\mathcal{M}$ comprises a state transition function τ on states over extended signature Σ_S with $\tau(S) = S + \Delta_{r_S}(S)$, where the rule r_S is defined as $raise(rule(\mathrm{val}_S(self)))$ over the signature $\Sigma_S = raise(signature(\mathrm{val}_S(self)))$.

In this definition we use extraction functions *rule* and *signature* defined on the tree representation of a sequential ASM in *self*. These are simply defined as $\mathit{signature}(t) = \mathit{subtree}(\mathbf{I}o.\mathit{root}(t) \prec_c o \wedge \mathit{label}(o) = \texttt{signature})$ and $\mathit{rule}(t) = \mathit{subtree}(\mathbf{I}o.\mathit{root}(t) \prec_c o \wedge \mathit{label}(o) = \texttt{rule})$.

3 The Capture of Genetic Algorithms

In [11] we explored genetic algorithms as a specific class of reflective algorithms, and demonstrated how the logic of reflective ASMs could be used to verify desirable properties of such algorithms. Programs in genetic programming are represented by their syntax trees, and in each macro-step a population of such programs is iteratively transformed into a new generation of programs by applying a fixed set of "genetic" operations until a chosen termination criterion is met. The rASM rule in Listing 1 formally specifies a run of a generic genetic algorithm as described in [6].

We assume that every state includes the following fixed background:

- a function *fitness* for measuring the fitness of individual programs in the population;
- a Boolean function *meet_term_crit* which returns true if the termination criteria is met;
- a set T of well formed syntax trees of programs;
- a function *prog* which maps each program tree in T to its corresponding expression, i.e. a well formed first-order term.

Parameters *max_d* and *init_method* specify the maximum depth and the method that must be used by the algorithm to create the initial random population of individual programs, i.e. the syntax trees of the programs. Here we consider the initialization methods "full" and "grow", but others are possible. In the "full" initialization method, the randomly generated syntax trees are full trees of depth *max_d*, where the internal nodes are labelled by functions and the leaf nodes are labelled by independent variables or constants. In the "grow" initialization method trees are also generated randomly, but they are not necessarily full, i.e. leaves can be at distance $< \mathit{max\_d}$ and not all leaves need to be at the same level. Another important control parameter is the population size *pop_size*.

```
GeneticAlgorithmRun(max_d, init_method, pop_size) =
if mode = init ∧ pop_size < card(gen(0)) then
  import x do
     AddFunc(x)
     AddUpdateRule(x)
     seq
     GenRndProg(x, max_d, init_method)
if mode = init ∧ pop_size = card(gen(0)) then
  n := 0
  mode := run
if mode = run then mode := eval
```

12 **if** $mode = \text{eval} \wedge \neg\exists x(x \in gen(n) \wedge meet\_term\_crit(result(x)))$ **then**
13 $\quad mode := \text{reprod}$
14 **if** $mode = \text{reprod} \wedge pop\_size < card(gen(n+1))$ **then**
15 $\quad$ **import** x **do**
16 $\qquad \textsc{AddFunc}(x)$
17 $\qquad \textsc{AddUpdateRule}(x)$
18 $\qquad$ **seq**
19 $\qquad$ **let** $gen\_op = select\_gen\_op()$ **in**
20 $\qquad\quad \textsc{GenerateOffspring}(x, gen\_op, max\_d)$
21 **if** $mode = \text{reprod} \wedge pop\_size \geq card(gen(n+1))$ **then**
22 $\quad n := n + 1$
23 $\quad mode := \text{run}$

Listing 1. Genetic Programming Algorithm

The specified algorithm works as follows. In the initial state ($mode = \text{init}$) it creates *pop_size* new nullary function symbols as well as *pop_size* update rule of the form $result(x) = fitness(prog(x))$, where x is one of the newly created function symbols. All these update rules are executed in parallel at a latter state when $mode = \text{run}$. Each state location for each newly created function symbol is further updated with a different randomly generated program tree and each of these trees is added to the initial generation of programs $gen(0)$. Once the initial generation of random programs has been produced, the algorithm switches to a run state and applies the (updated) rule corresponding to that in line 11 of Listing 1. Note that at this point this does *not* only mean to execute the update $mode := \text{eval}$, but also all the updates to *result* with the fitness value of the programs generated so far. The rule was augmented with those updates when the algorithm was in mode init. In the eval mode the algorithm simply checks whether the termination criteria has been met by some of the generated programs. If not, then the algorithm moves to the reprod mode, where a new generation of program is produced by applying different genetic operations to the programs of the current generation. The mechanism is similar to that used to produce the initial generation of programs, except that now the new programs are not generated randomly. The process continues in the same way until the termination criteria is met.

The sub-rules AddFunc and AddUpdateRule in Listings 2 and 3 apply reflection to add a new function symbol to the signature and to add an update sub-rule to the main rule of the rASM, respectively.

$\textsc{AddFunc}(x) =$
let $s = \mathbf{I}o.(root(self) \prec_c o \wedge label(o) = \texttt{signature})$ **in**
$s \xleftarrow{}{}^{right\_extend}\ label\_hedge(\texttt{func}, \langle x\rangle\langle 0\rangle)$

Listing 2. Add a new function symbol of arity 0 to the signature

$\textsc{AddUpdateRule}(x) =$
let $r = \mathbf{I}o.\exists o_1 o_2 o_{31} o_{32} o_{33} o_4 (root(self) \prec_c o_1 \wedge label(o_1) = \texttt{rule} \wedge o_1 \prec_c o_2$
$\quad \wedge o_2 \prec_c o_{31} \wedge \forall z(z \not\prec_s o_{31}) \wedge o_{31} \prec_s o_{32} \wedge o_{32} \prec_c o_{33} \wedge o_{33} \prec_c o_4 \wedge$
$\quad label(o_4) = \texttt{bool} \wedge o_4 \prec_s o)$ **in**

```
r ⇐^{right_extend} label_hedge(update, func⟨result⟩term⟨x⟩term⟨fitness(prog(x))⟩)
```

Listing 3. Add update rule to the body of the if-rule in Line 13 of Listing 1

The subrules GenRndProg and GenerateOffspring used in Listing 1, respectively, specify the generation of a random program and the creation of a new program (offspring) by applying the operations of reproduction, crossover and mutation. Another common type of operation is that of architecture-altering, but we omit that to simplify the presentation. We can specify these subrules by using the tree algebra of rASMs to create and manipulate syntax trees of ASM rules. These subrules can be specified as in Listing 4 and 5, respectively.

```
GenRndProg(x, max_d, init_method) =
if init_method = full then
  choose y ∈ T with y ∉ gen(0) ∧ depth(y) = max_d ∧ full_tree(y) do
    cur_gen(0, y) := true
    x := y
if init_method = grow then
  choose y ∈ T with y ∉ gen(0) ∧ depth(y) ≤ max_d do
    cur_gen(0, y) := true
    x := y
```

Listing 4. Generate Random Program

```
GenerateOffspring(x, gen_op, max_d) =
if gen_op = reproduction then
  choose y by fitness_prob_dist(y) with y ∈ gen(n) do
    gen(n + 1, y) := true
    x := y
if gen_op = mutation then
  choose y by fitness_prob_dist(y) with y ∈ gen(n) do
    choose z with z ∈ nodes_of(y) do
      choose w ∈ T with depth(w) + level(y) ≤ max_d do
        let y' = subst_tt(y, z, w) in
          gen(n + 1, y') := true
          x := y'
if gen_op = crossover then
  import y do
    AddFunc(y)
    AddUpdateRule(y)
    seq
    choose z_1 by fitness_prob_dist(z_1) with z_1 ∈ gen(n) do
    choose z_2 by fitness_prob_dist(z_2) with z_2 ∈ gen(n) ∧ z_2 ≠ z_1 do
      choose w_1 with w_1 ∈ nodes_of(z_1) do
      choose w_2 with w_2 ∈ nodes_of(z_2) do
        let z'_1 = subst_tt(z_1, w_1, subtree(w_2)) in
          gen(n + 1, z'_1) := true
          x := z'_1
```

```
let  z'_2 = subst_tt(z_2, w_2, subtree(w_1))  in
  gen(n + 1, z'_2) := true
  y := z'_2
```

Listing 5. Generate a new offspring

4 Recombinative Simulated Annealing

Parallel recombinative simulated annealing is a very popular genetic algorithm, which fine-tunes the general procedure described in the previous section[3] [7]. In this algorithm the genetic operations include mutation, crossover and Boltzmann trials, the latter ones controlled by a "temperature" value T. The algorithm can be defined informally as follows:

A) Initialise the temperature value *Temp* for Boltzmann trials to a sufficiently high value.
B) As in the generic case for genetic algorithms (cf. Listing 1) create *pop_size* new nullary function symbols as well as *pop_size* update rules to evaluate the newly created functions.
C) Update the newly created functions with randomly generated program trees, and add each of these trees to the initial population of programs.
D) Run each of the n programs in the population, get the corresponding fitness value, and check if it meet the termination criterion. If yes, terminate. Otherwise continue.
E) Randomly choose $n/2$ pairs of programs in the population and generate for each such pair two children using a recombination operator such as crossover followed by a neighborhood operator such as mutation. Then run the two children program and obtain their fitness values. Execute Boltzmann trials between children and parents, and overwrite parents with the winner.
F) Lower the Boltzmann temperature *Temp* and iterate the execution of E.

The algorithm for parallel recombinative simulated annealing is specified by the rASM in Listing 6, which is almost the same as the specification in Listing 1 with the modification that we use *Temp* and $\Delta temp$ as additional input values for the Boltzmann temperature and the decrement (used in F). The temperature value *Temp* is used in the subrule GenerateOffspring described in Listing 7. The requirements for the background remain the same as in the generic case except that we employ in addition a function *Boltzmax_dis* to generate new offsprings with Boltzmann distribution, which is used to decide whether the children or the parents will survive for the next generation. We use *max_d* and *init_method* to represent the maximum depth and the method for the creation of an initial random syntax trees.

[3] We dispense here with a proof that the rASM specification below satisfies the criteria required for ASM refinements [3].

```
ParRecombSimAnnealing(max_d, init_method, pop_size, Temp, Δtemp) =
if  mode = init ∧ pop_size < card(gen(0))  then
  import  x  do
     AddFunc(x)
     AddUpdateRule(x)
     seq
     GenRndProg(x, max_d, init_method)
if  mode = init ∧ pop_size = card(gen(0))  then
  n := 0
  mode := run
if  mode = run  then  mode := eval
if  mode = eval ∧ ¬∃x(x ∈ gen(n) ∧ meet_term_crit(result(x)))  then
  mode := reprod
if  mode = reprod ∧ pop_size < card(gen(n + 1))  then
  import  x, y  do
     AddFunc(x)
     AddUpdateRule(x)
     AddFunc(y)
     AddUpdateRule(y)
     seq
     GenerateOffspring(x, y, max_d, Temp)
  Temp := Temp − Δtemp
if  mode = reprod ∧ pop_size ≥ card(gen(n + 1))  then
  n := n + 1
  mode := run
```

Listing 6. Parallel Recombinative Simulated Annealing

```
GenerateOffspring(x, y, max_d, Temp) =
choose  z_1  with  z_1 ∈ gen(n)  do
choose  z_2  with  z_2 ∈ gen(n) ∧ z_2 ≠ z_1  do
  choose  w_1  with  w_1 ∈ nodes_of(z_1)  do
  choose  w_2  with  w_2 ∈ nodes_of(z_2)  do
     let  z'_1 = subst_tt(z_1, w_1, subtree(w_2))  in
     let  z'_2 = subst_tt(z_2, w_2, subtree(w_1))  in
        choose  n_1  with  n_1 ∈ nodes_of(z'_1)  do
        choose  n_2  with  n_2 ∈ nodes_of(z'_2)  do
           choose  w'_1 ∈ T  with  depth(w'_1) + level(z'_1) ≤ max_d  do
           choose  w'_2 ∈ T  with  depth(w'_2) + level(z'_2) ≤ max_d  do
              let  z''_1 = subst_tt(z'_1, n_1, w'_1)  in
              let  z''_2 = subst_tt(z'_2, n_2, w'_2)  in
              (x, y) := Boltzmax_dis(result(z_1), result(z_2), result(z''_1), result(z''_2))
```

Listing 7. Generate a new offspring

All these parameters are updated in parallel in states with "mode = run", where each function symbol will be updated with a different random program tree and added to the initial generation of the program gen(0). In evaluation mode, i.e. when “mode=eval” holds, the algorithm checks the termination criterion. If the termination condition is not met, mode will change to “reprod” causing the algorithm to proceed in reproduction mode and continue the previous process until termination.

The rules AddFunc, AddUpdateRule and GenRndProg are the same as in the generic case. The rule GenerateOffspring refines the generic version using the function *Boltzmax_dis*, in which **choose** rules are used to find a program in the current generation with probability based on fitness [12].

5 Software to Hardware Binding using Reflection

Industrial-scale reverse engineering is a serious problem, with estimated losses for the industry at 6,4 billion dollars in Germany alone[4]. A closer look to the problem, shows that typically the main effort needed to steal the intellectual property of companies producing machines controlled by software, resides on replicating hardware, since software can often be copied verbatim with no reverse engineering effort required. In this section we use rASMs to formally model the approach recently proposed in [8] to attack this problem.

The idea behind the copy protection described in [8] is to “glue” a program P to an specific machine M. More concretely, they propose to subtly change P into a (reflective) program P' which will turn itself into P at run time, only if it is run in the target machine M. If P' is executed in a machine M' other than M (even if M' is a clone of M), it will then behave incorrectly, i.e., differently than P. Clearly, for this approach to work, the changes that P' needs to make to its code to become P at run time need to be well protected. This can be achieved by making these changes dependent on physically unclonable properties of the target machine M, via a physically unclonable function (PUF).

Let us first illustrate this approach with a simple example. Consider the ASM specification in [2] (see Chapter 2, Section 2.1) of a one-way traffic light control algorithm. The proper behaviour of this algorithm is defined by the ASM rule in Listing 8. With a few subtle changes we can modify this rule so that it defines a different (incorrect) behaviour. An example of this is shown in Listing 9. This latter ASM rule defines an abstract executable program that still runs, but do not behave as required. For instance, if lights 1 and 2 are both in stop-mode and it is the turn of light 2 to switch to go-mode (see line 12-13 in Listing 8), the incorrect specification in Listing 9 determines that light 1 switch to go-mode instead. Notice that in this example we simply scramble the updates of *phase* in lines 6, 8, 13 and 15 (cf. Listing 8 and 9). In general, there is no restriction to the type of changes that one can do to best achieve the software protection goals.

```
1 1WayStopGoLight =
```

[4] VDMA Product Piracy 2022 (`https://www.vdma.org/documents/34570/51629660/VDMA+Study+Product+Piracy+2022_final.pdf`). Last accessed: 30/01/2023.

```
2  if phase ∈ {Stop1Stop2, Go1Stop2} and Passed(phase) then
3     StopLight(1) := ¬StopLight(1)
4     GoLight(1) := ¬GoLight(1)
5     if phase = Stop1Stop2 then
6        phase := Go1Stop2
7     else
8        phase := Stop2Stop1
9  if phase ∈ {Stop2Stop1, Go2Stop1} and Passed(phase) then
10    StopLight(2) := ¬StopLight(2)
11    GoLight(2) := ¬GoLight(2)
12    if phase = Stop2Stop1 then
13       phase := Go2Stop1
14    else
15       phase := Stop1Stop2
```

Listing 8. 1Way Traffic Light: Correct Specification

```
1  Incorrect1WayStopGoLight =
2  if phase ∈ {Stop1Stop2, Go1Stop2} and Passed(phase) then
3     StopLight(1) := ¬StopLight(1)
4     GoLight(1) := ¬GoLight(1)
5     if phase = Stop1Stop2 then
6        phase := Stop2Stop1
7     else
8        phase := Stop1Stop2
9  if phase ∈ {Stop2Stop1, Go2Stop1} and Passed(phase) then
10    StopLight(2) := ¬StopLight(2)
11    GoLight(2) := ¬GoLight(2)
12    if phase = Stop2Stop1 then
13       phase := Go1Stop2
14    else
15       phase := Stop2Stop1
```

Listing 9. Way Traffic Light: Incorrect Specification

Now we need to expand the rule in Listing 9 with reflective behaviour, so that it reverses itself back to the rule in Listing 8 whenever it is executed in the correct machine/hardware. That is, if the program executes in the target machine, it needs to update appropriately the relevant subtrees representing the update rules in lines 6, 8, 13 and 15 in the location *self* (i.e., the location that contains the self-representation of the ASM). It is key in this schema to protect the required tree replacement operations, so that an attacker cannot easily determine the correct program with certainty. The answer is to use an encoding that depends on a PUF. In particular, the approach proposed in [8] uses uses rowhammer –a fault injection bug in DRAM modules that allows unprivileged malicious actors to flip bits in physical memory– as an intrinsic PUF [9,1]. Since the bit flips (from 0 to 1 or vice-versa, depending on the memory cell type) produced by rowhammer are due to unavoidable variances in the manufacturing process of the DRAM chips, the set of bit flips and the rows that contain these bits constitute a unique and unclonable identifier for these chips. In this paper we do not dig deeper into how these type of PUF can be implemented, since it is accessory to the goal of modelling the protection mechanism at the level of abstraction considered in this paper. Furthermore, other kinds of PUFs could be used in practice without fundamentally changing the method.

Thus, we simply assume here that there is a PUF *swap* which takes as input a binary string b and returns as output a possibly different binary string b'. For each binary string b in the domain of *swap*, we assume that we know the corresponding $b' = swap(b)$ in the target machine/hardware M. Since *swap* is a PUF, $swap(b)$ will be interpreted at runtime as b' only if the program is executed in M. Otherwise, *swap* is assumed to be interpreted as the identity function. The *swap* function is treated at this specification level as a monitored functions (see [3]), i.e., a dynamic functions which is read but not updated by the ASM specification and directly updatable only by the environment. In latter refinement steps, this function can be specified by means of an ASM description of the specific PUF used in the implementation. For instance, b' could be the result of applying a rowhammer exploit to flip some bits of b as in [8]. In addition, we assume a function *bin* which encodes syntax trees of ASM rules as binary strings. The encoding function *bin* must satisfy the following constraint: $swap(bin(t)) = bin(t')$ whenever the algorithm is run on M and the "incorrect" rule represented by the tree t needs to be swapped by the "correct" rule represented by t'.

We can now proceed to complete our example of copy protection for the algorithm in Listing 8. A protected version of this algorithm is shown in Listing 10. In the *programUpdate* mode (which we assume for every initial state), the algorithm replaces the update rules in lines 24, 26, 31 and 33 using the PUF *swap* and the encoding *bin*. If the algorithm is executed in the target machine M, this will result in these subrules being changed to the updates in lines 6, 8, 13 and 15 from Listing 8. After this first step, the algorithm enters the *execution* mode and works as intended. In case the algorithm is execute in a machine other than M, then the result is that the rules in the *execution* mode will remain the same as in Listing 8 and the algorithm will behave incorrectly.

1 PROTECTED1WAYSTOPGOLIGHT =
2 **if** $mode = programUpdate$ **then**
3   **let**
    $n_0 = \mathbf{I} o_1 . \exists o_0 o_2 o_3 (root(self) \prec_c^+ o_0 \prec_c o_1 \prec_c o_2 \prec_c o_3 \wedge label(o_0) = \texttt{rule} \wedge$
4         $label(o_1) = \texttt{update} \wedge label(o_2) = \texttt{term} \wedge label(o_3) = \mathit{Stop2Stop1} \wedge$
5         $\exists o_4 (o_4 \prec_s o_0 \wedge label(o_4) = \texttt{bool}))$
6     $n_1 = \mathbf{I} o_0 . \exists o_1 o_2 (root(self) \prec_c^+ o_0 \prec_c o_1 \prec_c o_2 \wedge label(o_0) = \texttt{update} \wedge$
7         $label(o_1) = \texttt{term} \wedge label(o_2) = \mathit{Stop1Stop2})$
8     $n_2 = \mathbf{I} o_0 . \exists o_1 o_2 (root(self) \prec_c^+ o_0 \prec_c o_1 \prec_c o_2 \wedge label(o_0) = \texttt{update} \wedge$
9         $label(o_1) = \texttt{term} \wedge label(o_2) = \mathit{Go1Stop2})$
10     $n_3 = \mathbf{I} o_1 . \exists o_0 o_2 o_3 (root(self) \prec_c^+ o_0 \prec_c o_1 \prec_c o_2 \prec_c o_3 \wedge label(o_0) = \texttt{rule} \wedge$
11         $label(o_1) = \texttt{update} \wedge label(o_2) = \texttt{term} \wedge label(o_3) = \mathit{Stop2Stop1} \wedge$

$\quad\quad\quad\quad \exists o_4(o_4 \prec_s o_0 \wedge \mathit{label}(o_4) = \texttt{rule}))$
$\quad$ **in**
$\quad\quad \mathit{self} \leftleftarrows^{subst_{tt}} n_0, \mathit{bin}^{-1}(\mathit{swap}(\mathit{bin}(\mathit{subtree}(n_0))))$
$\quad\quad \mathit{self} \leftleftarrows^{subst_{tt}} n_1, \mathit{bin}^{-1}(\mathit{swap}(\mathit{bin}(\mathit{subtree}(n_1))))$
$\quad\quad \mathit{self} \leftleftarrows^{subst_{tt}} n_2, \mathit{bin}^{-1}(\mathit{swap}(\mathit{bin}(\mathit{subtree}(n_2))))$
$\quad\quad \mathit{self} \leftleftarrows^{subst_{tt}} n_3, \mathit{bin}^{-1}(\mathit{swap}(\mathit{bin}(\mathit{subtree}(n_3))))$
$\quad \mathit{mode} := \mathit{execution}$
if $\mathit{mode} = \mathit{execution}$ **then**
$\quad$ **if** $\mathit{phase} \in \{\mathit{Stop1Stop2}, \mathit{Go1Stop2}\}$ **and** $\mathit{Passed}(\mathit{phase})$ **then**
$\quad\quad \mathit{StopLight}(1) := \neg \mathit{StopLight}(1)$
$\quad\quad \mathit{GoLight}(1) := \neg \mathit{GoLight}(1)$
$\quad\quad$ **if** $\mathit{phase} = \mathit{Stop1Stop2}$ **then**
$\quad\quad\quad \mathit{phase} := \mathit{Stop2Stop1}$
$\quad\quad$ **else**
$\quad\quad\quad \mathit{phase} := \mathit{Stop1Stop2}$
$\quad$ **if** $\mathit{phase} \in \{\mathit{Stop2Stop1}, \mathit{Go2Stop1}\}$ **and** $\mathit{Passed}(\mathit{phase})$ **then**
$\quad\quad \mathit{StopLight}(2) := \neg \mathit{StopLight}(2)$
$\quad\quad \mathit{GoLight}(2) := \neg \mathit{GoLight}(2)$
$\quad\quad$ **if** $\mathit{phase} = \mathit{Stop2Stop1}$ **then**
$\quad\quad\quad \mathit{phase} := \mathit{Go1Stop2}$
$\quad\quad$ **else**
$\quad\quad\quad \mathit{phase} := \mathit{Stop2Stop1}$

Listing 10. Way Traffic Light: Protected Specification

In PROTECTED1WAYSTOPGOLIGHT we specify once (at the beginning of the run) the reflective behaviour required to make the algorithm run as intended (provided it is executed in the target machine). We could generalize this to a schema where each execution step is preceded by a (reflective) program update step, in which the correction is done (if necessary). That is, the program update step determined by the PUF can be done on demand. One could use this globally as in PROTECTED1WAYSTOPGOLIGHT, i.e. do the program update at once, or locally, i.e. the program is updated on demand. Each of the update-execution steps could be followed by restoring the incorrect code, so that an attacker that can perform a dynamic analysis of the algorithm in the target machine will still have a hard time determining the necessary changes to make the algorithm behave correctly in a cloned machine. Regardless, a static analysis as well as a dynamic analysis in a hardware other than the one associated to the PUF, will not reveal the correct code. The general strategy for software to hardware binding using rASMs together with PUFs is formally specified by the ground model in Listing 11.

PROTECTEDPROGRAMRULE $=$
if $\mathit{mode} = \mathit{init}$ **then**
$\quad \mathit{program} := \mathbf{I}o_0.\exists o_1 o_2 O_3(\mathit{root}(\mathit{self}) \prec^+ o_2 \wedge \mathit{label}(o_2) = \texttt{bool}) \wedge o_2 \prec_c o_3 \wedge$
$\quad\quad\quad\quad \mathit{label}(o_3) = \text{“}\mathit{mode} = \mathit{execution}\text{”} \wedge o_2 \prec_s o_0 \wedge \mathit{label}(o_0) = \texttt{rule})$
$\quad \mathit{mode} := \mathit{changePoints}$

```
if mode = changePoints then
   nodes := selectNodes(subtree(program))
   mode := programUpdate
if mode = programUpdate then
   forall n ∈ nodes do
      self ⇇^{subst_tt} n, bin^{-1}(swap(bin(subtree(n)))
      initialSubRule(n) := subtree(n)
   mode := execution
if mode = execution then
   ProgramRule
   if executionDone then
      mode := reverseChanges
if mode = reverseChanges then
   forall n ∈ nodes do
      self ⇇^{subst_tt} n, initialSubRule(n)
   nodes := ∅
   mode := changePoints
```

Listing 11. Software to Hardware Binding: Ground Model

At this point we could already exploit the logic for rASMs developed in [11] to express some desired properties of this model. For instance, we could express that unless the algorithm is in *execution* or *reverseChanges* mode, the content of *self* is the same as in the initial state. Thus, an attacker that is performing a dynamic analysis of the algorithm can only see changes to the ProgramRule if she/he observes the content of *self* in a state where mode equals *execution* or *reverseChanges*, and the program is executing in the target machine.

$$\varphi \equiv \mathit{mode} = \mathit{init} \rightarrow \forall x X (x \in \mathbb{N}^{+} \wedge \text{r-upd}(x, X) \wedge [X]\mathit{mode} \neq \mathit{execution} \wedge$$

$$[X]\mathit{mode} \neq \mathit{reverseChanges} \rightarrow \mathit{self} = [X]\mathit{self})$$

Likewise, we can express concrete properties over the algorithm in Listing 10. Let us assume that the model has a protected and static location *targetMachine* with Boolean value true iff the algorithm is executing in the target machine/hardware. Then one can for instance express that the algorithm behaves as intended with respect to the update in Line 24 , whenever it is in *execution* mode in the target machine.

$$\psi \equiv \mathit{mode} = \mathit{execution} \wedge \mathit{targetMachine} \rightarrow$$

$$\forall x_0 x_1 x_2 x_3 y_0 y_1 (\mathit{root}(\mathit{self}) \prec_c^{+} x_0 \prec_c x_1 \prec_c x_2 \prec_c x_3 \wedge y_0 \prec_s x_0 \wedge y_0 \prec_c y_1$$

$$\mathit{label}(x_0) = \texttt{rule} \wedge \mathit{label}(x_1) = \texttt{update} \wedge \mathit{label}(x_2) = \texttt{term} \wedge$$

$$\mathit{label}(x_3) = \mathit{Go1Stop2} \wedge \mathit{label}(y_0) = \texttt{bool} \wedge \mathit{label}(y_1) = \text{“}\mathit{phase} = \mathit{Stop1Stop2}\text{”})$$

Similarly, we could check for instance whether the algorithm behaves in the way it is expected with respect to this update rule whenever it is executed in a machine other than the target one. This is important for instance to ensure that the incorrect behaviour is controlled and cannot produce harm.

$$\psi \equiv mode = execution \wedge \neg targetMachine \rightarrow$$

$$\forall x_0 x_1 x_2 x_3 y_0 y_1 (root(self) \prec_c^+ x_0 \prec_c x_1 \prec_c x_2 \prec_c x_3 \wedge y_0 \prec_s x_0 \wedge y_0 \prec_c y_1$$

$$label(x_0) = \texttt{rule} \wedge label(x_1) = \texttt{update} \wedge label(x_2) = \texttt{term} \wedge$$

$$label(x_3) = Stop2Stop1 \wedge label(y_0) = \texttt{bool} \wedge label(y_1) = \text{“}phase = Stop1Stop2\text{”})$$

We have shown how the general reflective strategy to bind software to hardware can be naturally modelled using rASM. We have also shown how we can rigorously express desired properties of this models using the logic for rASMs. It is of course also possible (and desirable) to apply standard techniques for ASM refinement and formally verify every step up to implementation. Indeed, the method as described in [8] uses the binary object code and not a high-level specification, but that doesn't change the essential idea that only at run time the incorrect fragments of the code are replaced by the correct ones. For instance, if we have an ASM rule "**if** <*cond*> **then** r_1 **else** r_2", then the implemented and compiled binary will have a subprogram to evaluate <*cond*> and jump-instructions to the code compiled from r_1 or r_2, respectively. Thus, the modification could be much more atomic changing only a single machine code instruction instead of the whole ASM rule. But using refinement we could create an ASM for the low-level code as well.

6 Concluding Remarks

In this article we explored the expressive power of reflective Abstract State Machines (rASMs). We demonstrated that all genetic algorithms are captured by rASMs, and provided an rASM specification of recombinative simulated annealing as a specific example. We further show that security methods for copy protection can also be supported and verified by using rASMs.

The examples exploit the full power of rASMs including parallelism, partial updates, choice and coupling with probability distributions. However, the behavioural theory of reflective algorithms so far only covers reflective sequential algorithms. In order to show that all reflective algorithms are captured by rASMs it will be necessary to extend the theories as already envisioned in [10].

References

1. Nikolaos Athanasios Anagnostopoulos, Tolga Arul, Yufan Fan, Christian Hatzfeld, André Schaller, Wenjie Xiong, Manishkumar Jain, Muhammad Umair Saleem, Jan Lotichius, Sebastian Gabmeyer, Jakub Szefer, and Stefan Katzenbeisser. Intrinsic run-time row hammer pufs: Leveraging the row hammer effect for run-time cryptography and improved security †. *Cryptogr.*, 2(3):13, 2018. `doi:10.3390/cryptography2030013`.

2. Egon Börger and Alexander Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018. `doi:10.1007/978-3-662-56641-1`.
3. Egon Börger and Robert Stärk. *Abstract State Machines*. Springer-Verlag, Berlin Heidelberg New York, 2003.
4. David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.
5. John H. Holland. Studying complex adaptive systems. *J. Syst. Sci. Complex.*, 19(1):1–8, 2006.
6. John R. Koza and Riccardo Poli. A genetic programming tutorial. In Graham Kendall Edmund K. Burke, editor, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 127–164. Springer, 2005.
7. Samir W. Mahfoud and David E. Goldberg. Parallel recombinative simulated annealing: A genetic algorithm. *Parallel Comput.*, 21(1):1–28, 1995.
8. Ruben Mechelinck, Daniel Dorfmeister, Stijn Volckaert, and Stefan Brunthaler. μGLUE: Efficient and scalable software to hardware binding using rowhammer. *Submitted for Publication*, 2023.
9. André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Intrinsic rowhammer pufs: Leveraging the rowhammer effect for improved security. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, pages 1–7. IEEE Computer Society, 2017. `doi:10.1109/HST.2017.7951729`.
10. Klaus-Dieter Schewe and Flavio Ferrarotti. Behavioural theory of reflective algorithms I: reflective sequential algorithms. *Sci. Comput. Program.*, 223:102864, 2022. `doi:10.1016/j.scico.2022.102864`.
11. Klaus-Dieter Schewe, Flavio Ferrarotti, and Senén González. A logic for reflective ASMs. *Sci. Comput. Program.*, 210:102691, 2021. `doi:10.1016/j.scico.2021.102691`.
12. Klaus-Dieter Schewe, Flavio Ferrarotti, Loredana Tec, and Qing Wang. Towards a behavioural theory for random parallel computing. In Christoph Beierle, Gerhard Brewka, and Matthias Thimm, editors, *Computational Models of Rationality, Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday*, pages 365–376. College Publications, 2016.
13. Klaus-Dieter Schewe and Qing Wang. Partial updates in complex-value databases. In *Information and Knowledge Bases XXII*, volume 225 of *Frontiers in Artificial Intelligence and Applications*, pages 37–56. IOS Press, 2011.
14. David Stemple, Leo Fegaras, Robin Stanton, Tim Sheard, Paul Philbrow, Richard Cooper, Malcolm P. Atkinson, Ron Morrison, Graham Kirby, Richard Connor, and Suad Alagic. Type-safe linguistic reflection: A generator technology. In *Fully Integrated Data Environments*, Esprit Basic Research Series, pages 158–188. Springer Berlin Heidelberg, 2000.