Monte Carlo Analysis of Security Protocols: Needham-Schroeder Revisited[*]

R. Grosu, X. Huang, S.A. Smolka, P. Yang
Department of Computer Science
State University of New York at Stony Brook
`grosu,xhuang,sas,pyang@cs.sunysb.edu`

Abstract

We apply Monte Carlo model checking to the Needham-Schroeder public key authentication protocol. The Monte Carlo approach uses random sampling of "lassos" (reachable cycles) to compute an estimate of the weighted expectation that a system S satisfies an LTL formula φ within a factor of $1 \pm \epsilon$ with probability at least $1 - \delta$. It does so using a number of samples N that is optimal to within a constant factor, and in expected time $O(N \cdot D)$ and expected space $O(D)$, where D is the recurrence diameter of the directed graph representing the product of S's state transition graph and the Büchi automaton for $\neg\varphi$. Our results indicate that Monte Carlo model checking can find attacks in security protocols like Needham-Schroeder when traditional model checkers fail due to state explosion; and that the weighted expectation that Needham-Schroder is attack-free increases linearly with the nonce range (number of rounds).

1 Introduction

The Needham-Schroeder public-key authentication protocol, first published in 1978 [16], initiated a large body of work on the design and analysis of cryptographic protocols. In 1995, Lowe published an attack on the protocol that had apparently been undiscovered for the previous 17 years [12]. The following year, he showed how the flaw could be discovered mechanically by model checking [13], and this has been followed by many papers on model checking and automated verification of similar protocols [15, 1, 8, 17, 7, 4, 14, e.g.].

In this paper, we describe how the recently discovered technique of *Monte Carlo model checking* [9] can be used for this purpose. The Monte Carlo approach uses random sampling of "lassos" (reachable cycles) to compute an estimate of the weighted expectation that a system S satisfies an LTL formula φ within a factor of $1 \pm \epsilon$ with probability at least $1 - \delta$. It does so using a number of samples N that is optimal to within a constant factor, and in expected time $O(N \cdot D)$ and expected space $O(D)$, where D is the diameter of the directed graph representing the product of S's state transition graph and the Büchi automaton for $\neg\varphi$ [18].

Our results will show that Monte Carlo model checking can find attacks in security protocols like Needham-Schroeder when traditional model checkers fail due to state explosion; and that the weighted expectation that Needham-Schroder is attack-free increases linearly with the nonce range (number of rounds).

The intent of the Needham-Schroeder protocol is to establish mutual authentication between principals A and B in the presence of an intruder who can intercept, delay, read, copy, and generate messages, but who does not know the private keys of the principals. The fragment of the protocol that is subject to the attack discovered by Lowe [12] is specified as follows:

$$\begin{array}{ll} 1. & A \rightarrow B : \{N_A, A\}_{K_B} \\ 2. & B \rightarrow A : \{N_A, N_B\}_{K_A} \\ 3. & A \rightarrow B : \{N_B\}_{K_B} \end{array}$$

Principal A begins by sending to B an encrypted message containing a nonce (a previously unused and

[*]R. Grosu and X. Huang were partially supported by the NSF Faculty Early Career Development Award CCR01-33583.

unpredictable identifier) N_A and its own identity. The encryption uses B's public key and can be decoded by B using its private key, but is indecipherable to all other participants.

B responds with a similar message to A, including the nonce received from A and a new nonce of its own, which is readable only by A. A examines the second message and concludes that it really is from B (since only B could have discovered the nonce N_A) and that it is not a replay (because N_A is current). It then returns nonce N_B to B under B's public key. On receipt of this message, B uses similar reasoning to determine that it really is from A, and is current, and so both principals have authenticated the other.

The flaw discovered by Lowe uses an interleaving of two runs of the protocol, as shown below. A initiates a run with principal I, who is corrupt (i.e., an intruder). I then initiates a run with B, purporting to be A and using the nonce provided by A. B replies with a message encrypted for A that I uses unchanged in a message of its own to A. A decrypts this to discover B's nonce (thinking it is I's) and sends it to I under I's public key. I now knows B's nonce and can complete its run with B. At this point, B believes it has authenticated A, but it is actually talking to I.

$$\begin{array}{lr@{\;:\;}l}
1a. & A \rightarrow I & \{N_A, A\}_{K_I} \\
1b. & I(A) \rightarrow B & \{N_A, A\}_{K_B} \\
2b. & B \rightarrow I(A) & \{N_A, N_B\}_{K_A} \\
2a. & I \rightarrow A & \{N_A, N_A\}_{K_A} \\
3a. & A \rightarrow I & \{N_B\}_{K_I} \\
3b. & I(A) \rightarrow B & \{N_B\}_{K_B}
\end{array}$$

Here, $I(A)$ indicates I masquerading as A, and the suffices a and b on the message numbers indicate which run of the protocol they belong to.

As shown by Lowe in [13], the protocol is easily fixed by including the identity of the responder (B) in the second message (preventing the replay of 2b in 2a).

$$2'. \quad B \rightarrow A : \{B, N_A, N_B\}_{K_A}$$

In order to demonstrate the effectiveness of Monte Carlo model checking at uncovering attacks in the presence of increasingly larger state spaces, we shall show how this technique can be applied to a specification of Needham-Schroeder that permits an arbitrary number of runs to be captured.

The rest of the paper develops along the following lines. Section 2 reviews `MC`^{2}, our Monte Carlo model-checking algorithm of [9]. Section 3 describes our encoding of Needham-Shroeder in our jMocha [2] implementation of `MC`^{2}. Section 4 summarizes our experimental results. Section 5 contains our conclusions and directions for future work.

2 Monte Carlo Model Checking

In this section, we review our randomized approach to LTL model checking presented in [9]. `MC`^{2}, our Monte Carlo model-checking algorithm, is based on the `DDFS` double depth-first search algorithm used in automata-theoretic model checking, and on the `OAA` optimal approximation algorithm of [6] for Monte Carlo estimation.

2.1 LTL Model Checking

LTL (Linear Temporal Logic) is a well-studied temporal logic for defining correctness properties of concurrent systems. The set of well-formed LTL formulas is constructed from a finite set of atomic propositions AP, the standard boolean connectives, and the temporal operators "neXt state" (`X`) and "Until" (`U`).

In automata-theoretic LTL model checking, the problem of deciding $S \models \varphi$, for system S and LTL formula φ is reduced to the *language emptiness problem* for finite automata over infinite words [18]. The reduction involves modeling S and $\neg\varphi$ as Büchi automata B_S and $B_{\neg\varphi}$, respectively, taking the product $B = B_S \times B_{\neg\varphi}$, and checking whether the language $L(B)$ of B is empty.[1]

Checking (non-)emptiness of $L(B)$ is equivalent to finding a strongly connected component of B that is reachable from an initial state and contains an accepting state. Due to the acceptance condition for Büchi automata, however, this reduces to finding a reachable accepting *cycle*. Looking for such a cycle is usually done by using the *double depth-first search* algorithm `DDFS` of [5, 10]. `DDFS` interleaves two depth-first searches `DFS1` and `DFS2`. When `DFS1` is ready to backtrack from an accepting state after completing the search of its successors, it starts `DFS2` in search of a cycle through this state. If `DFS2` fails to find such a cycle, it resumes `DFS1` from the point it was interrupted.

One can avoid the explicit construction of B_S by generating its initial and successor states on demand and performing the test for acceptance symbolically. This *on-the-fly* approach considerably improves the space requirements of `DDFS`, since it constructs only the reachable part of B_S.

2.2 Optimal Monte Carlo Estimation

Many engineering and computer-science applications require the computation of the mean value μ_Z for a random variable Z distributed in $[0, 1]$. When an exact computation of μ_Z proves intractable, being, for example, NP-hard, Monte Carlo methods are often used to compute an (ϵ, δ)-approximation of this quantity. The main idea is to use N independent random variables (or samples) $Z_1, \ldots, Z_N$ identically distributed according to Z with mean μ_Z, and to take $\widetilde{\mu}_Z = (Z_1 + \ldots + Z_N)/N$ as the approximation of μ_Z.

An important issue in such an approximation scheme is determining the value for N. Let σ_Z^2 be the variance of Z. The *generalized zero-one estimator theorem* of [6] states that if N is proportional to

$$\Upsilon = \begin{cases} 4\ln(2/\delta)/\mu_Z\epsilon & \text{if } \sigma_Z^2 \leq \epsilon\mu_Z \\ 4\ln(2/\delta)/\mu_Z\epsilon^2 & \text{otherwise} \end{cases}$$

then $\widetilde{\mu}_Z$ approximates μ_Z with absolute error ϵ and with probability $1 - \delta$. More precisely:

$$\mathbf{Pr}[\mu_Z(1-\epsilon) \leq \widetilde{\mu}_Z \leq \mu_Z(1+\epsilon)] \geq 1 - \delta$$

Thus, if the variance of Z is small relative to $\epsilon\mu_Z$, then $\frac{1}{\epsilon}$ fewer samples are needed to compute an (ϵ, δ)-approximation of μ_Z. In practice, this observation can result in substantial computational savings.

To apply the generalized zero-one estimator theorem, one requires the values of the unknown quantities μ_Z and σ_Z^2. This problem can be circumvented by finding an upper bound κ for ρ_Z/μ_Z, where $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$, and using κ to compute N. Finding a tight upper bound is however in most cases very difficult, and a poor choice of κ leads to a prohibitively large value for N.

To avoid the problem encountered with the generalized zero-one estimator theorem, the authors of [6] have proposed the `OAA` optimal approximation algorithm. `OAA` uses the outcomes of previous experiments to decide when to stop iterating, a technique known as *sequentail analysis*. `OAA` relies critically on the *Stopping Rule Algorithm* (SRA), which has the following property: when $\mathbf{E}[Z] = \mu_Z > 0$ and $\Sigma_i Z_i \geq \Upsilon$, the expected number of samples taken by SRA with respect to Z on input ϵ and δ is given by the second clause of the defining equation for Υ. Essentially, SRA computes a running sum of the Z_i, terminating when this sum reaches Υ. By also keeping track of how many samples Z_i are taken in the process, the desired value of N is determined.

The `OAA` algorithm consists of three steps, the first of which calls the SRA algorithm with parameters $(\sqrt{\epsilon}, \delta/3)$ to get an estimate $\widehat{\mu}_Z$ of μ_Z. The choice of parameters is based on the assumption that $\rho_Z = \epsilon\mu_Z$,

[1]The rationale behind this reduction is as follows:

$$S \models \varphi \text{ iff } L(B_S) \subseteq L(B_\varphi) \text{ iff } L(B_S) \cap \overline{L(B_\varphi)} = \emptyset \text{ iff } L(B_S) \cap L(B_{\neg\varphi}) = \emptyset \text{ iff } L(B_S \times B_{\neg\varphi}) = \emptyset$$

and ensures that SRA takes $3/\epsilon$ less samples than would otherwise be the case. The second step uses $\widehat{\mu}_Z$ to get an estimate of $\widehat{\rho}_Z$. The third step uses $\widehat{\rho}_Z$ to get the desired value $\widetilde{\mu}_Z$. Should the assumption $\rho_Z = \epsilon\mu_Z$ fail to hold, the second and third steps will compensate by taking an appropriate number of additional samples. As shown in [6], `OAA` runs in an expected number of experiments that is within a constant factor of the minimum expected number.

2.3 The Monte Carlo Model-Checking Algorithm

Our `MC`^{2} algorithm uses the `OAA` algorithm of [6] to compute an (ϵ, δ)-approximation of a certain weighted expectation that $S \models \varphi$. The *samples* taken by `MC`^{2} are the reachable cycles or "lassos" of a Büchi automaton B that the `DDFS` algorithm has been desisgned to search for.[2] Should B be the product automaton $B_S \times B_{\neg\varphi}$, then a lasso containing a final state of B (an "accepting lasso") can be interpreted as a *counter-example* to $S \models \varphi$. A lasso of B is sampled via a random walk through B's transition graph, starting from a randomly selected initial state of B.

Definition 1 (Sample space) Given a Büchi automaton B, a finite run $s_0x_0 \ldots s_nx_ns_{n+1}$ of B is called a *lasso* if $s_0 \ldots s_n$ are pairwise distinct and $s_{n+1} = s_i$ for some $0 \leq i \leq n$. The *sample space* U of B is the set of all lassos of B. □

Definition 2 (Run probability) The *probability* $\mathbf{Pr}[\sigma]$ of a finite run $\sigma = s_0x_0 \ldots s_{n-1}x_{n-1}s_n$ of a Büchi automaton B is defined inductively as follows: $\mathbf{Pr}[s_0] = k^{-1}$ if $|Q_0| = k$ and $\mathbf{Pr}[s_0x_0 \ldots s_{n-1}x_{n-1}s_n] = \mathbf{Pr}[s_0x_0 \ldots s_{n-1}] \cdot \pi[s_{n-1}x_{n-1}s_n]$ where $\pi[s\,x\,t] = m^{-1}$ if $(s, x, t) \in \delta$ and $|\delta(s)| = m$. □

Proposition 1 (Probability space) *Given a Büchi automaton B, the pair $(\mathcal{P}(U), \mathbf{Pr})$ defines a discrete probability space.*

Definition 3 (Random variable) The *random variable* Z associated with the probability space $(\mathcal{P}(U), \mathbf{Pr})$ of a Büchi automaton B is defined as follows: $\mathbf{Pr}[Z = 0] = \sum_{\lambda_a \in U} \mathbf{Pr}[\lambda_a]$ and $\mathbf{Pr}[Z = 1] = \sum_{\lambda_n \in U} \mathbf{Pr}[\lambda_n]$ where λ_a is an accepting lasso and λ_n is a non-accepting lasso. □

The *expectation* (or weighted mean) $\mu_Z = 0 \cdot \mathbf{Pr}[Z = 0] + 1 \cdot \mathbf{Pr}[Z = 1]$ of Z is equal to $\mathbf{Pr}[Z = 1]$. It provides a measure of the number of counter-examples (accepting lassos) in B, weighted by their probability. Since an exact computation of μ_Z is often intractable due to state explosion, we compute an (ϵ,δ)-approximation $\widetilde{\mu}_Z$ of μ_Z using the `OAA` algorithm. We then use $\widetilde{\mu}_Z$ to derive a Monte Carlo decision procedure we call `MC`^{2} (*Monte Carlo Model Checking*) for the LTL model-checking problem. `MC`^{2} works as follows: (1) Take independent random samples (lassos) Z_i and Z'_i, each identically distributed according to Z with mean μ_Z as required by `OAA`. (2) If an accepting lasso is encountered, break and return the lasso as a counterexample. (3) If all samples are non-accepting, conclude that μ_Z is 1 with error margin ϵ and confidence ratio δ.

Our use of `OAA` thus yields a one-sided-error decision procedure for the LTL model-checking problem as `MC`^{2} correctly decides false if $\widetilde{\mu}_Z < 1$. A similar approach is possible for the DNF satisfiability problem: use the Monte Carlo algorithm of [11] to estimate the ratio of satisfying truth assignments for a given DNF formula and decide true if $\widetilde{\mu}_Z > 0$. This yields a randomized algorithm for DNF SAT belonging to the complexity class RP: if the correct answer is YES, then it returns YES with probability at least $\frac{1}{2}$, and if the correct answer is NO, then it always returns NO. In contrast, `MC`^{2} can be seen as belonging to the class co-RP (see also Theorem 1): if the correct answer is NO, then it returns NO with probability at least $\frac{1}{2}$ (assuming $\delta \leq \frac{1}{2}$), and if the correct answer is YES, then it always returns YES.

The pseudo-code for `MC`^{2} is now given, where `acc(s,B)=(s` $\in$ `F)`, `rInit(B)=random(S`$_0$`)`, `rNext(s,B)=`τ`.t` and τ`=random({(s,`α`,t) | (s,`α`,t)` $\in \delta$`})`. The main routine consists of a single statement in which the `OAA` algorithm is called with parameters ϵ, δ, and the *random accepting cycle variable* (`RACV`) routine, which generates the random variables (samples) Z_i and Z'_i used in `OAA` on demand as follows. A random lasso

[2]We assume without loss of generality that every state of a Büchi automaton B has at least one outgoing transition, even if this transition is a self-loop.

```
MC^2 algorithm
input:   Büchi automaton B = (Σ, Q, Q_0, δ, F);
input:   Error margin ε and confidence ratio δ with 0 < ε ≤ 1 and 0 < δ ≤ 1.
output:  Either counterexample or estimation $\widetilde{\mu}_Z$ with $\mathbf{Pr}[\mu_Z(1-\epsilon) \le \widetilde{\mu}_Z \le \mu_Z(1+\epsilon)] \ge 1-\delta$

(1)  try { $\widetilde{\mu}_Z$ = OAA(ε, δ, RACV(B)); return $\widetilde{\mu}_Z$; }  catch(e) { return e; }
```

```
RACV algorithm
input:   Büchi automaton B;
output:  Samples a random cycle of B; throws HashTbl if cycle is accepting; returns 1 otherwise.

(1)  s := rInit(B); i := 1; f := 0;
(2)  while (s ∉ HashTbl) {
(3)    HashTbl(s) := i;
(4)    if (acc(s,B)) f := i;
(5)    s := rNext(B,s); i := i+1; }
(6)  if (HashTbl(s) ≤ f) throw(HashTbl) else return 1;
```

is generated using the *randomized init* (`rInit`) and *randomized next* (`rNext`) routines. To determine if the generated lasso is accepting, we store the index `i` of each encountered state `s` in `HashTbl` and record the index of the most recently encountered accepting state in `f`. When we find a cycle, i.e., the state returned by `rNext(M,s)` is in `HashTbl`, we check if `HashTbl(t)` $\le$ `f`; the cycle is an accepting cycle if and only if this is the case.

As with `DDFS`, given a succint representation S of a Büchi automaton B, one can avoid the explicit construction of B by generating random states `rInit(B)` and `rNext(s,B)` on demand and performing the test for acceptance `acc(s,B)` symbolically. In the next section we present such a succint representation and show how to efficiently generate random initial and successor states.

Theorem 1 (`MC`^{2} correctness) Given a Büchi automaton B, error margin ϵ, and confidence ratio δ, $\widetilde{\mu}_Z$, the (ϵ,δ)-approximation of μ_Z computed by `MC`^{2} is such that if $\widetilde{\mu}_Z < 1$ then $L(B) \neq \emptyset$, and if $\widetilde{\mu}_Z = 1$ then the wighted expectation μ_Z that $L(B) = \emptyset$ satisfies $\mathbf{Pr}[1/\mu_Z - 1 \ge \epsilon] \le \delta$.

`MC`^{2} is very efficient in both time and space. The *recurrence diameter* of a Büchi automaton B is the longest initialized loop-free path in B. Also, observe that the number of samples taken by `OAA` when all Z_i, Z'_i return 1 has a well-defined value for each (ϵ, δ) pair.

Theorem 2 (`MC`^{2} expected complexity) Let B be a Büchi automaton, D its recurrence diameter and N be the number of samples taken by `OAA` when all Z_i and Z'_i return 1, for a given ϵ, δ. Then, `MC`^{2} takes expected time $O(N \cdot D)$ and uses expected space $O(D)$.

`MC`^{2} can also be run in "estimator mode", where it does not halt upon finding a counter-example but rather continues sampling until the computation of $\widetilde{\mu}_Z$ is completed. By virtue of its reliance on the `OAA` algorithm, `MC`^{2} in estimator mode may not terminate if the number of initialized non-accepting cycles in B is less than Υ_1. Should this not be the case, however, `MC`^{2} provides an estimate of how "false" is the judgement $S \models \varphi$, a useful statistical measure.

3 Implementation

We have implmented the `DDFS` and `MC`^{2} algorithms as an extension to jMocha [2], a model checker for synchronous and asynchronous concurrent systems specified using *reactive modules* [3]. An LTL formula $\neg\varphi$ is specified in our extension of jMocha as a pair consisting of a reactive module monitor and a boolean formula defining its set of accepting states. By selecting the new enumerative or randomized LTL verification option, one can check whether $S \models \varphi$: jMocha takes their composition and applies $\mathtt{MC}^2$ on-the-fly to check for accepting lassos.

Using jMocha, we specified the Needham-Schroeder protocol as a reactive module such that all communications between the principals go through the intruder and the intruder behaves according to the Dolev-Yao model: it can perform normal communication and intercept, overhear, or fake messages.

A nonce is uniquely represented as a pair consisting of a value and an id. Initially, the value is set to 0 and subsequently incremented by 1 each time principal A or B generates a nonce. This ensures that A and B generate fresh nonces each time one is needed. We have also explicitly specified the range of nonces. The intruder I generates a nonce using the jMocha*nondet* command, which randomly generates an integer value within this range. This technique allows the intruder to generate a fresh nonce, or possibly a nonce used previously by A or B.

Having nonce ranges allows us to model multiple runs of the protocol in that they determine the maximum number of runs for which the protocol can execute. The larger the nonce range, the more runs the protocol can execute. Our Monte Carlo model checker terminates a random walk if: (i) all the nonces in the range are used up; (ii) an attack is found; or (iii) the system enters an invalid state (for instance, the intruder fakes a message and sends it to a principal, but the principal discards the message since the message is not the one it expects).

The authenticity property is specified as correspondence assertion [19], i.e., a pair, consisting of a reactive module monitor and a state predicate. When A initiates a run with B, it emits an event $\mathtt{rq_B}$ which triggers a change in the state `ms` of the monitor from normal `NRM` to requested `RQS`. When B commits to communicate with A, it emits an event $\mathtt{cm_B}$, which changes the monitor state back to `NRM` if the current state is `RQS`, and to committed `CMM` otherwise. An attack on B is then expressed with the predicate `ms` = `CMM`, which means that there exists a path along which B commits to A without a corresponding initiation.

A fragment of the Needham-Schroeder reactive module is shown below. It consists of a collection of typed variables partitioned into *external* (input), *interface* (output) and *private*. For this example, the interface variables `id`, `pk`, `fr`, `to`, `va`, `rq` and `cm` represent the network state and denote the id, public key, from, to, value, request and commit, respectively. The pair $(\mathtt{id_1}, \mathtt{va_1})$ stores a unique nonce and the pair $(\mathtt{id_2}, 0)$ stores an id. The private variables `pc`, `li`, `lv`, `uv` `lp`, `ac` and `r2` constitute a A and B's state and denote the program counter, last identifier, last value, used value, last public key, action and random selector, respectively. The process states `IDL`, `INT` and `RES` denote idle, initiated and responded, respectively.

```
type IdType is {Z,ID_A,ID_B,ID_I}; ProcState is {IDL,INI,RES}; Value is (0..60);
module NeedhamSchroeder is
  interface id_1,id_2,pk,fr,to:IdType; va_1,va_2:Value; rq_A,rq_B,cm_A,cm_B:event;
  private pc_A,pc_B:ProcState; li_A,li_B,lp_A,lp_B:IdType; lv_A,lv_B:Value; ac:(0..28); r2:(0..1);
  atom Random controls r2
    initupdate
      [] true -> r2':= nondet;
  atom NextAction controls ac reads pc_A,pc_B,fr,to,pk
    init
      [] true -> ac':= 0;
    update
      ...
      // message #1
      [] fr = Z & to = Z & pc_A = IDL -> ac':= 1;
      [] fr = Z & to = Z & pc_B = IDL -> ac':= 9;
      // about to commit authentication
      [] fr = ID_I & to = ID_A & pc_A = RES & pk = ID_A -> ac':= 4;
      [] fr = ID_I & to = ID_B & pc_B = RES & pk = ID_B -> ac':= 8;
      ...
  atom Send
```

```
    controls id_1,id_2,va_1,va_2,pk,pc_A,pc_B,fr,to,li_A,li_B,lv_A,lv_B,lp_A,lp_B,uv_A,uv_B,rq_A,rq_B,cm_A,cm_B
    reads id_1,id_2,va_1,va_2,pk,fr,to,li_A,li_B,lv_A,lv_B,lp_A,lp_B,uv_A,uv_B
    awaits ac,r2
    init
      [] true -> id'_1:=Z; id'_2:=Z; va'_1:=0; va'_2:=0; pk':=Z; fr':=Z; to':=Z; pc'_A:= IDL; pc'_B:=IDL;
            li'_A:=Z; li'_B:=Z; lv'_A:=0; lv'_B:=0; lp'_A:=Z; lp'_B:=Z; uv'_A:=0; uv'_B:=0;
    update
      ...
      [] ac'=1 & r2'=1 -> id'_1:= ID_A; va'_1:=uv_A; id'_2:=ID_A; va'_2:=0; pk':=ID_B;
            lp'_A:=ID_B; pc'_A:=INI; li'_A:=ID_A; lv'_A:=uv_A; fr':=ID_A; to':= ID_I; rq_B!
      [] ac'=8 & li_B=id_1 & lv_B=va_1 & lp_B=ID_A -> id'_1:=Z; id'_2:=Z; va'_1:=0; va'_2:=0;
            pk':=Z; lp'_B:=Z; li'_B:=Z; lv'_B:=0; pc'_B:=IDL; fr' :=Z; to' :=Z; uv'_B:=uv'_B+1; cm_B!
      ...
```

Variables change their values in a sequence of rounds. The first is an *initialization* round; the subsequent are *update* rounds. Initialization and updates of controlled (interface and private) variables are specified by *actions* defined as a set of *guarded parallel assignments*. Moreover, controlled variables are partitioned into *atoms*: each variable is initialized and updated by exactly one atom.

The initialization round and all update rounds are divided into subrounds, one for the environment and one for each atom A. In an A-subround of the initialization round, all variables controlled by A are initialized simultaneously, as defined by an initial action. In an A-subround of each update round, all variables controlled by A are updated simultaneously, as defined by an update action.

In a round, each variable x has two values: the value at the beginning of the round, written as x and called the *read value*, and the value at the end of the round written as x' and called the *updated value*. *Events* are modeled by toggling boolean variables. For example $\texttt{rq}_\texttt{B}? \overset{\text{def}}{=} \texttt{rq}'_\texttt{B} \neq \texttt{rq}_\texttt{B}$ and $\texttt{rq}_\texttt{B}! \overset{\text{def}}{=} \texttt{rq}'_\texttt{B} := \neg\texttt{rq}_\texttt{B}$. If a variable x controlled by an atom A depends on the updated value y' of a variable controlled by atom B, then B has to be executed before A. We say that A *awaits* B and that y is an awaited variable of A. The await dependency defines a partial order $\succ$ among atoms.

```
type MonitorState is {NRM,RQS,CMM}
module Monitor is
  external rq,cm:event; interface ms:MonitorState;
  atom Watch controls ms reads ms awaits cm,rq
    init
      [] true -> ms' := NRM;
    update
      [] ms = NRM & rq?  -> ms' := RQS;
      [] ms = RQS & cm?  -> ms' := NRM;
      [] ms = NRM & cm?  -> ms' := CMM;
module System is NeedhamSchroeder || Monitor[rq,cm,ms:=rq_A,cm_A,ms_A] || Monitor[rq,cm,ms:=rq_B,cm_B,ms_B]
predicate NoCommitWoRequest is (ms_A ≠ CMM & ms_B ≠ CMM)
judgment NoFlaw is System |= NoCommitWoRequest
```

Operators on modules include *renaming*, *hiding* of output variables, and *parallel composition*. The latter is defined only when the modules update disjoint sets of variables and have a joint acyclic await dependency. In this case, the composition takes the union of the private and interface variables, the union of the external variables (minus the interface variables), the union of the atoms, and the union of the await dependencies. For example, the module `System` is the parallel composition of the Needham-Schroeder module and the renamed monitors for `A` and `B`, respectively. The safety property is given by the predicate `NoCommitWoRequest`.

rNext algorithm
input: `Reactive module M; Current state s;`
output: `Random next state s.all′.`

```
(1) s.extl′ := random(Q.M.extl);
(2) for all (A ∈ ≻^L_M) {
(3)   for (m := |A.upd|; m ≥ 0; m--) {
(4)     i := random(m);
(5)     if (A.upd(i).grd(s)) break else remove(A.upd,i); }
(6)   if (m = 0) s.ctrl′ := s.ctrl; else s.ctrl′ := random(A.upd(i).ass(s)); }
(7) return s′;
```

A feature of our MC^2 implementation in jMocha is that the next state `s′ = rNext(s,M)` of `M` along a random walk in search of an accepting lasso is generated randomly both for the external variables `M.extl` and for the controlled variables `M.ctrl`. For the external variables we randomly generate a state `s.extl′` in the set of all input valuations `Q.M.extl`. For the controlled variables we proceed for each atom `A` in the linear order $\succ^L_{\mathtt{M}}$ compatible with $\succ_{\mathtt{M}}$ as follows: first we randomly choose a guarded assignment `A.upd(i)` with true guard `A.upd(i).grd(s)`, where `i` is less than the number `|A.upd|` of guarded assignments in `A`; then we randomly generate a state `s.ctrl′` among the set of all states possibly returned by its parallel (nondeterministic) assignment `A.upd(i).ass(s)`. If no guarded assignment is enabled we keep the current state `s.ctrl`. The routine `rInit` is implemented in a similar way.

4 Experimental Results

We compared the performance of `DDFS` and MC^2 on Needham-Shroeder using our implementation of these algorithms in jMocha. Specifically, we checked whether the Reactive-Module `System`, encoding the Needham-Schroeder protocol, composed with the monitors satisfies the predicate `NoCommitWoRequest`.

The results are shown in Table 1(a) and were obtained on a PC equipped with an Athlon 2100+ MHz processor and 1GB RAM running Linux 2.4.18 (Fedora Core 1). The `nonce` column represents the range of nonces; this determines the maximal number of runs the protocol can execute. For `DDFS`, the `time` column represents the time in seconds taken to detect an attack and the `entries` column represents the corresponding number of entries in the hash table. For MC^2, the `time` column represents the time in seconds required by MC^2 to run in estimator mode on all N random samples (lassos). The MC^2 `exp` column is the time when the first counter-example (attack) is expected to appear, calculated as MC^2 (`time`)/(`counter`) where `counter` is the number of counter-examples. The `avg` column represents the average length of an MC^2 lasso which is the same as the average number of entries in the hash table. For MC^2, we used $\epsilon = \delta = 0.1$.

	DDFS		MC^2		
nonce	time	entries	time	exp	avg
(0..1)	1	31	20	1	12
(0..4)	1	607	33	2	29
(0..8)	2	2527	34	9	30
(0..20)	11	24031	34	12	30
(0..32)	32	85279	70	24	30
(0..36)	46	118111	141	37	30
(0..60)	–	oom	4200	467	29

	MC^2		
nonce	satisf.	counter.	$\widetilde{\mu}_Z$
(0..1)	2915	171	0.9445
(0..4)	2955	18	0.9939
(0..8)	2969	4	0.9986
(0..20)	2970	3	0.9989
(0..32)	6288	3	0.9995
(0..36)	12975	3	0.9997
(0..60)	194937	9	0.9999

Table 1: (a) Time and space requirements for `DDFS` and MC^2. (b) Variation of $\widetilde{\mu}_Z$ for MC^2.

The *quantitative* information computed by MC^2 is shown in Table 1(b) where the `satisf` column is the number of lassos that do not contain attacks, the `counter` column is the number of lassos that are attacks (counter-examples), and $\widetilde{\mu}_Z$ is the (ϵ, δ)-approximation of μ_Z, the weighted expectation that an attack does not occur. Observe that there are far more lassos that do not contain attacks than those containing attacks. For instance, when the range of nonces is 60, there are only 9 lassos containing an attack among 194946 lassos traversed. Note also that the probability of an attack decreases as the nonce range increases. This observation is fairly obvious in retrospect, but to our knowledge has not been reported previously in the model checking literature.

Table 1(a) shows that when the number of states is small, the `DDFS` checker, which uses a form of partial-order reduction, is faster than $\texttt{MC}^2$. When the number of states is large, however, $\texttt{MC}^2$ outperforms `DDFS`. For example, when the nonce range is (0..32), the expected time that the first counter-example appears is 24 seconds while the enumerative model checker takes 32 seconds. When the nonce range is (0..60), the enumerative model checker runs out of memory whereas $\texttt{MC}^2$ finds the first counter-example in expected time 467 seconds. Moreover, after 4200 seconds and 194946 lassos traversed, $\texttt{MC}^2$ is also able to provide the desired *quantitative* information.

5 Conclusions

We applied Monte Carlo model checking to the Needham-Schroeder authentication protocol, a well-established benchmark in the field of security-protocol analysis. Our results indicate that the Monte Carlo technique may be more effective than traditonal approaches in discovering attacks, expecially in terms of scalability.

Further experimentation is required to draw any definitive conclusions about the power of Monte Carlo model checking in analyzing secutiry protocols, for example, on other protocol benchmarks, and in comparison with other analysis techniques, such as those cited in the Introduction.

As future work, we also plan to improve the time and space efficiency of our jMocha implementation of $\texttt{MC}^2$ by "compiling" it into a BDD representation. More precisely, we plan to encode the current state, hash table, and guarded assignments of each atom in a reactive module as BDDs, and to implement the next-state computation and the containment (in the hash table) check as BDD operations.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] R. Alur, L. de Alfaro, R. Grosu, T. A. Henzinger, M. Kang, C. M. Kirsch, R. Majumdar, F. Mang, and B. Y. Wang. jMocha: A model checking tool that exploits design structure. In *Proceedings of the 23rd international conference on Software engineering*, pages 835–836. IEEE Computer Society, 2001.

[3] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, July 1999.

[4] B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium*, volume 2477 of *Lecture Notes on Computer Science*, pages 242–259, Madrid, Spain, September 2002. Springer Verlag.

[5] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

[6] P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for Monte Carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.

[7] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand Spaces: Proving security protocol correct. *Journal of Computer Security*, 7:191–230, 1999.

[8] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *Transactions on Software Engineering*, pages 550–571, 1997.

[9] R. Grosu and S. A. Smolka. Monte Carlo model checking. Technical report, Department of Computer Science, SUNY Stony Brook, 2004. `http://www.cs.sunysb.edu/~sas/papers/GS04.pdf`.

[10] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. pages 23–32, 1996.

[11] R. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10:429–448, 1989.

[12] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, pages 131–133, 1995.

[13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[14] W. Mackw M. Kurkowski. Using backward strategy to the Needham-Schroeder public key protocol verification. *Artificial Intelligence and Security in Computing Systems*, pages 249–259, 2003.

[15] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In *ESORICS: European Symposium on Research in Computer Security.* LNCS, Springer-Verlag, 1996.

[16] R. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, (12):993–999, 1978.

[17] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[18] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[19] Woo and Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.