Parallel Maude-NPA for Cryptographic Protocol Analysis

Canh Minh Do, Adrián Riesco, Santiago Escobar, and Kazuhiro Ogata

Abstract—Maude-NPA is a formal verification tool for analyzing cryptographic protocols in the Dolev-Yao strand space model modulo an equational theory defining the cryptographic primitives. It starts from an attack state to find counterexamples or conclude that the attack concerned cannot be conducted by performing a backward narrowing reachability analysis. Although Maude-NPA is a powerful analyzer, its running performance can be improved by taking advantage of parallel and/or distributed computing when dealing with complex protocols whose state space is huge. This paper describes a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption at each layer in Maude-NPA are conducted in parallel. A tool supporting the parallel version has been implemented in Maude with a master-worker model. We report on some experiments of various kinds of protocols that demonstrate that the tool can increase the running performance of Maude-NPA by 44% on average for complex case studies in which the number of states located at each layer is considerably large.

Index Terms—Meta-interpreters, Maude, Master-worker Model, Parallel Maude-NPA, Cryptographic Protocol Analysis.

1 Introduction

Cryptographic protocols (or security protocols), such as Transport Layer Security (TLS) [2], [3], are extremely important so as to implement secure, safe, and reliable communication over an open network, such as the Internet. It is also extremely hard to design such protocols and detect flaws lurking in them [4]. Some protocols designed by security experts had flaws and it took time to discover them after their publication. For example, Lowe found a flaw [5] in the Needham-Schroeder public key (NSPK) authentication protocol 17 years later since NSPK was designed and published by Needham and Schroeder [6], security experts. TLS is an improved version of SSL, which is commonly used in securing HTTPS connections. Since SSL version 1.0 was first introduced in 1994, many attacks were quickly discovered, and so SSL versions 2.0 and 3.0 were then proposed in 1995 and 1996, respectively. By 2015, SSL 3.0 became deprecated and TLS was then used as an upgrade version because of some attacks [7], such as the POODLE attack. Recently, in response to the threats posed by quantum computing, Amazon Web Services has proposed a hybrid key exchange in TLS 1.3 [8] to establish a shared secret that can resist quantum attacks in the future. Therefore, techniques and tools that help researchers and engineers find out flaws lurking in cryptographic protocols and/or formally verify that such protocols enjoy some desired properties are indispensable. Tools dedicated to formal verification of such protocols have been developed. Among them are Maude-NPA [9], Athena [10], ProVerif [11], Avispa [12], CL-Atse [13], Scyther [14], Tamarin [15], AKISS [16], DEEPSEC [17], Verifpal [18], and CPSA [19]. Because the running performance is crucial, such tools adopt several optimization techniques (e.g., the partial order reduction [20]) like those used by general-purpose model checkers, such as Spin [21] and NuSMV [22]. ~~Except for DEEPSEC, which supports protocols with a bounded number of sessions, none of the dedicated formal verification tools for cryptographic protocols have been parallelized~~A few tools have been parallelized to improve their performance, such as CL-Atse, Tamarin, AKISS, and DEEPSEC. Among them, only DEEPSEC has well-documented how the parallelization has been done [17], while others have not discussed it in detail regarding our investigation. For example, Tamarin only briefly tells us how to set the number of threads with some parameters for parallelization in their manual.

Maude-NPA is a formal verification tool for analyzing cryptographic protocols. It uses a backward narrowing reachability analysis modulo an equational theory and the Dolev-Yao strand space model [23], [24], which gives intruders capable of intercepting, modifying, and injecting messages to impersonate other protocol principals. Narrowing is a generalization of term rewriting that allows logical variables in terms and replaces pattern matching by unification. Hence, it supports symbolic execution. The backward narrowing reachability analysis starts from a final insecure state, an attack state, to determine whether it is reachable from an initial state, which has no further backward steps. If so, the attack concerned from the attack state can be conducted for the protocol under verification; otherwise, the attack cannot. The advantage of Maude-NPA is that it supports protocols with an unbounded session model thanks to symbolic execution, and different equational theories; as a counterpart, these theories often lead to a bigger state space that requires more time to con-

The present paper is an extended and improved version of the workshop paper [1].

- *C. M. Do and K. Ogata are with School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Japan. E-mail: {canhdo,ogata}@jaist.ac.jp*
- *Adrián Riesco is with Universidad Complutense de Madrid, Spain. E-mail: ariesco@ucm.es*
- *Santiago Escobar is with Universitat Politècnica de València, Spain. E-mail: sescobar@upv.es*

Manuscript received April 19, 2005; revised August 26, 2015.

duct formal verification. Although some techniques were devised to reduce the state space and improve the running performance, such as grammar-based techniques, giving priority to input messages in strands, early detection of inconsistent states (never reaching an initial state), a relation of transition subsumption (to discard transitions and states already being processed in another part of the search space), and the super lazy intruder (to delay the generation of substitution instances as much as possible) [25], [26], the state space explosion problem is inevitable in some cases, degrading the running performance. Therefore, it is worth improving the running performance of Maude-NPA by taking advantage of parallel and/or distributed computing. To the best of our knowledge, Maude-NPA has been fully optimized, meaning that it would be impossible to improve its running performance without relying on parallelization. Although parallelization is one possible optimization technique to improve the running performanceof a tool, however, Parallelization is one of the main streams to improve running performance. However, it is really tough to improve the running performance of any tool that has already adopted many optimization techniques in a serial way, such as Maude-NPA. Moreover, it requires a deep and careful analysis of how a tool works in detail in order to parallelize it.

Maude-NPA uses a breadth-first search (BFS) to explore the reachable state space. Given a set of states in layer l (or depth l from an attack state, where the attack state is located at layer 0), for each state in the set, Maude-NPA performs the backward narrowing just by one step to obtain its successor states in layer $l+1$, which is referred to as step (1). The backward narrowing for each given state from the set of states can be conducted independently, which opens an opportunity for parallelization so as to improve the running performance of Maude-NPA. In addition, as soon as the successor states at each layer are obtained from step (1), Maude-NPA conducts the transition subsumption, which is referred to as step (2). The transition subsumption can be regarded as the partial order reduction for narrowing-based state exploration, to remove states that are implied by either other states in the successor states or visited states (history states) from the set of successor states.

We have parallelized step (1) in our previous work [1]. In the present paper, we describe how to parallelize step (2) as well to improve the running performance of Maude-NPA even more. Our parallel version of Maude-NPA is called Par-Maude-NPA. To distinguish Par-Maude-NPA from the version described in [1], it may be called Par-Maude-NPA-2, while the version described in [1] may be called Par-Maude-NPA-1. Maude-NPA is implemented in Maude [27] and so are Par-Maude-NPA-1 and Par-Maude-NPA-2, where Maude is one direct successor language of OBJ3 [28], an algebraic specification language, and based on rewriting logic [29] as its theoretical foundation. Both Par-Maude-NPA-1 and Par-Maude-NPA-2 use a master-worker model. Par-Maude-NPA-1 uses Maude sockets to transmit data between the master and a worker. Maude sockets work well for Par-Maude-NPA-1 because only a small amount of data is transmitted between the master and a worker in step (1). However, it is necessary to transmit large data between the master and a worker in step (2) for parallelization. Thus, Par-Maude-NPA-2 uses meta-interpreters, a new feature of Maude, instead of Maude sockets.

Meta-interpreters can be run as a separate process to handle jobs independently and processes can communicate with each other using Unix domain sockets that create filesystem objects to communicate between processes on the same host with no IP address required. Meanwhile, Maude sockets use TCP/IP sockets that require a unique IP address and a port to communicate between two parties in the same host or different hosts. The paper [30] demonstrates that Unix domain sockets are about two times faster than TCP/IP sockets. Besides, it is mandatory to convert data to a string representation before sending them over the network with Maude sockets, while it is not with meta-interpreters. Therefore, the use of meta-interpreters is efficient in Par-Maude-NPA-2.

In summary, the contributions of the present paper are as follows:

- We propose how to parallelize step (2) as well as step (1) of Maude-NPA.
- We implement Par-Maude-NPA-2 in Maude using meta-interpreters and its source code is publicly available at the following website: https://github.com/canhminhdo/parallel-maude-npa.
- We conduct many case studies demonstrating that Par-Maude-NPA-2 improves running performances of both Maude-NPA (about 44%) and Par-Maude-NPA-1 (about 23%) on average for complex cryptographic protocols used for experiments.

The rest of the paper is organized as follows. Section 2 mentions some preliminaries in which narrowing is described. Section ?? describes how to parallelize an application in Maude with object-oriented systems and meta-interpreters. Section 3 describes the overview of Maude-NPA. Section 4 describes our parallel version of Maude-NPA and a tool that supports it. Section 5 reports on some experimental results. Section 6 mentions some existing work. Section 7 gives some discussion. Finally, Section 8 concludes the paper together with some future directions. Readers may skip Sections 2, ??, 3 and go to Section 4 in order to understand how the parallelization has been performed.

2 PRELIMINARIES

Maude is a declarative language and high-performance tool that focuses on simplicity, expressiveness, and performance to support the formal specification and analysis of concurrent programs/systems in rewriting logic. The language can directly specify order-sorted equational logics and rewriting logic [29], and the tool provides several formal analysis methods, such as reachability analysis and LTL model checking. This section gives the syntax of the Maude language in a nutshell (see [27] for more details) and describes how narrowing works with an example.

Functional modules

A functional module $\mathcal{M}$ specifies an order-sorted equational logic theory (Σ, E) with the syntax: **fmod** $\mathcal{M}$ **is** (Σ, E) **endfm**, where Σ is an order-sorted signature and E is the

collection of equations in the functional module. (Σ, E) may contain a set of declarations as follows:

- importations of previously defined modules (**protecting** ... or **extending** ... or **including** ...)
- declarations of sorts (**sort** s . or **sorts** $s\ s'$.)
- subsort declarations (**subsort** $s < s'$.)
- declarations of function symbols (**op** f : $s_1\ \ldots\ s_n \rightarrow s$ [$att_1\ \ldots\ att_k$] .)
- declarations of variables (**vars** $v\ v'$.)
- unconditional equations (**eq** $t = t'$.)
- conditional equations (**ceq** $t = t'$ **if** $cond$.)

where $s, s_1, \ldots, s_n$ are sort names, v, v' are variable names, t, t' are terms, $cond$ is a conjunction of equations (i.e., $t = t'$ and/or t => t'), and $att_1, \ldots att_k$ are equational attributes. Equations are used as *equational rules* to perform the simplification in which instances of the lefthand side pattern that match subterms of a subject term are replaced by the corresponding instances of the righthand side. The process is called *term rewriting* and the result of simplifying a term is called its *normal form*.

System modules

A system module $\mathcal{R}$ specifies a rewrite theory (Σ, E, R) with the syntax: **mod** $\mathcal{R}$ **is** (Σ, E, R) **endm**, where Σ and E are the same as those in an equational theory and R is the collection of rewrite rules in the system module. (Σ, E, R) may contain all possible declarations in (Σ, E) and rewrite rules in R as follows:

- unconditional rewrite rules (**rl** [$label$] : u => v .)
- conditional rewrite rules (**crl** [$label$] : u => v **if** $cond$.)

where $label$ is the name of a rewrite rule, u, v are terms, and $cond$ is a conjunction of equations and/or rewrites (e.g., t => t'). Rewrite rules are also computed by rewriting from left to right modulo the equations in the system module and regarded as *local transition rules*, making possible many state transitions from a given state in a concurrent system.

Narrowing

Narrowing is a generalization of term rewriting that allows logical variables in terms and replaces pattern matching by unification. Let us use a classical example in the Maude community to describe how narrowing works. The formal definition of narrowing can be found in [1]. The following system module specifies a concurrent machine to buy cakes (c) and apples (a) with dollars ($) and quarters (q). We suppose that a cake costs a dollar (see the rewrite rule labeled as buy-c below) while an apple costs three quarters (the rewrite rule buy-a). The machine only allows buying cakes and apples with dollars. However, the machine can change four quarters into a dollar (the equation change).

```
mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  subsort Money Item < Marking .
---
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm
                                id: empty] .
  op __ : Marking Marking -> Marking [assoc comm
                                      id: empty] .
  op <_> : Marking -> State .
~~ ops $ q : -> Coin .~~
~~ ops c a : -> Item .~~
---
  ops $ q : -> Coin .     ops c a : -> Item .
  var M : Marking .
---
  rl [buy-a] : < M $ > => < M a q > [narrowing] .
  rl [buy-c] : < M $ > => < M c > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```

where the __ operator is associative and commutative and has an identity element empty, the <_> operator specifies the machine state, the narrowing attributes denote the rewrite rules used for narrowing, and the variant attribute denotes the equation used only for variant-based equational unification [31] in narrowing.

Let us consider a term < M1 > as an initial state that only contains a variable M1 of the sort Money. There would be several traces from the initial state by using narrowing. At each narrowing step, we must choose which subterm of the subject term being concerned, which rule of the specification, and which instantiation on the variables of the subterm and the left-hand side of the rewrite rule (or which unifier~~(or substitution)~~/substitution of the subterm and the left-hand side of the rewrite rule) are going to be considered. Note that only rewrite rules with the narrowing attribute are considered and only equations with the variant attribute are used to decide the unification problem modulo the equations. Each narrowing step applied to a given state produces a new branch in the reachability tree. For example, for each rewrite rule of the machine, there is only one unifier that makes the initial state equal to the left-hand side of the rewrite rule. Therefore, we can only obtain the following two narrowing steps, generating only two successor states from the initial state, by performing narrowing just by one step as follows:

$$\texttt{< M1 >} \rightsquigarrow_{\sigma_1,\ \texttt{buy-a}} \texttt{< a q M2 >}$$
$$\texttt{< M1 >} \rightsquigarrow_{\sigma'_1,\ \texttt{buy-c}} \texttt{< c M2' >}$$

where M2 and M2' are variables of sort Money and the substitutions are σ_1 = {M1 $\mapsto$ $ M2, M $\mapsto$ M2} and σ'_1 = {M1 $\mapsto$ $ M2', M $\mapsto$ M2'} with the rewrite rules buy-a and buy-c, respectively. Note that M in the substitutions is the variable used in the left-hand side of the rewrite rules. If we take the successor state < a q M2 > and perform two more consecutive narrowing steps, it makes one trace taking us to the state < a a c q M4 >, which also contains a variable M4 of the sort Money. The narrowing sequence associated to the state is as follows:

$$\texttt{< M1 >} \rightsquigarrow_{\sigma_1,\ \texttt{buy-a}} \texttt{< a q M2 >} \rightsquigarrow_{\sigma_2,\ \texttt{buy-c}} \texttt{< a c q M3 >} \rightsquigarrow_{\sigma_3,\ \texttt{buy-a}} \texttt{< a a c q M4 >}$$

where M3 and M4 are variables of the sort Money and the substitutions are σ_2 = {M2 $\mapsto$ $ M3, M $\mapsto$ a q M3 } and σ_3 = {M3 $\mapsto$ q q q M4, M $\mapsto$ a c M4} with the rewrite rules buy-c and buy-a, respectively. In the third narrowing step, when we apply the substitution σ_3, the instance of < a c q M3 > is < a c q q q q M4 >, while

the instance of the left-hand side of the rewrite rule `buy-a` is `< a c M4 $ >`. The two instances are actually equal thanks to the commutative property and the equation `change`. Therefore, the rewrite rule `buy-a` modulo the equational theory is used to obtain the state `< a a c q M4 >`. By using narrowing, we can solve the reachability problem $St \rightsquigarrow_{R,E} St'$ where St and St' are patterns (terms that may have variables) of the sort `State` such that some conditions are satisfied, and R, E are the rewrite rules and equations in the specification, respectively.

3 Object-oriented Systems and Meta-interpreters

Maude supports formally specifying concurrent systems in an object-oriented style where concurrent object systems are modeled by a set of *objects* that can interact with each other by sending and receiving *messages*. Moreover, meta-interpreters are a new feature in Maude that are external objects representing external entities with independent Maude interpreters, which have their own module and view databases and maintain their own states. Each meta-interpreter is run as a separate process that can send and receive messages to/from other objects from which each job can be handled by a meta-interpreter independently. Therefore, we use object-oriented programming and meta-interpreters in Maude to build the parallel version of Maude-NPA in this present paper.

Object-oriented systems

We can formally specify concurrent systems as object-oriented systems using the module `CONFIGURATION` in Maude. The state of an object-oriented system is called a *configuration* that is a multiset of objects and messages whose sort is `Configuration`. Configurations are denoted by an empty syntax (`none`) and a juxtaposition operator (`__`), which is declared associative and commutative and has `none` as its identity. For object syntax, there are four sorts introduced: `Oid` (object identifiers), `Cid` (class identifiers), `Attribute` (an attribute of an object), and `AttributeSet` (multisets of attributes). In this syntax, an *object* is a term of the sort `Object` with the following form:

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ att_1\texttt{,}\ \ldots\texttt{,}\ att_n\ \texttt{>}$$

where `O` is the object's identifier of the sort `Oid`, `C` is the object's class identifier of the sort `Cid`, and $att_1, \ldots, att_n$ are the object's attributes of the sort `AttributeSet`. A *message* is a term of the sort `Msg` where the declaration defines the syntax of the message $m(v_1, \ldots, v_n)$ and the sorts $s_1, \ldots, s_n$ of its parameters $v_1, \ldots, v_n$ as follows:

$$\texttt{op}\ m\ \texttt{:}\ s_1\ \ldots\ s_n\ \texttt{-> Msg [ctor] .}$$

Although messages do not have a fixed syntactic form, we follow a convention that the first and second arguments of a message are the identifiers of its destination and source objects, respectively. For example, let us specify a client-server communication in which there are several clients and servers, and the status of each server or client is either *idle* or *busy*. Each server can have many clients but each client can communicate with only one server. If a client C is *idle*, the client sends a request `N`, a natural number, to a server S and becomes *busy* (see the rewrite rule `req` below). If the server S is *idle*, then S receives the request (message), becomes *busy*, and returns `N + 1` to C as a message (the rewrite rule `repl`), meaning that the server increments `N`. Suppose that only the server knows how to increment a natural number. A *busy* client can receive an answer and become *idle* (the rewrite rule `recv`), and a *busy* server can become *idle* at any time (the rewrite rule `idle`). The system can be specified by the following system module:

```
mod CLIENT-SERVER is
  protecting NAT .
  including CONFIGURATION .
--
  sorts Status .
---
  ops Client Server : -> Cid [ctor] .
  ops idle busy : -> Status [ctor] .
--
    op status :_ : Status -> Attribute [ctor] .
  op val :_ : Nat -> Attribute [ctor] .
  op to :_ : Oid -> Attribute [ctor] .
  op m : Oid Oid Nat -> Msg [ctor] .
---
  vars N N' : Nat .
  vars C S : Oid .
----
--
    rl [req]: < C : Client | status : idle, val : N,
      to : S >
  => < C : Client | status : busy, val : N,
      to : S > m(S, C, N) .
  rl [repl]: < S : Server | status : idle >
      m(S, C, N)
  => < S : Server | status : busy >
      m(C, S, (N + 1)) .
--
    rl [recv]: < C : Client | status : busy, val : N,
      to : S > m(C, S, N')
  => < C : Client | status : idle, val : N',
      to : S > .
  rl [idle]: < C : Server | status : busy >
  => < C : Server | status : idle > .
endm
```

where the natural number that needs to be incremented by a server is stored in the `val` attribute of a client.

Let us suppose that there is a server `s` communicating with two clients `c1` and `c2` in the client-server system. Initially, the status of each `s`, `c1`, and `c2` is *idle*, the values of the `val` attributes of `c1` and `c2` are 3 and 4, respectively, and the value of the `to` attribute of each `c1` and `c2` is `s`. The initial state (referred to as `init`) and the object identifiers (`s`, `c1`, and `c2`) are defined in the following system module:

```
mod CLIENT-SERVER-TEST is
```

```
  extending CLIENT-SERVER .
--
  ops s c1 c2 : -> Oid .
  op init : -> Configuration .
--
  eq init = < s : Server | status : idle >
--
    < c1 : Client | status : idle, val : 3, to : s >
--
    < c2 : Client | status : idle, val : 4, to : s > .
endm
```

Given init, the rewrite rule req can be applied to the term expressing it at two positions, meaning that there are at least two execution traces that start with init, making many possible traces in a concurrent system.

Meta-interpreters

We can work with meta-interpreters using the module META-INTERPRETER in Maude that contains several sorts, constructors, a built-in object identifier interpreterManager, and a collection of command and response messages. There are some key messages as follows:

- The createInterpreter message is sent to the object interpreterManager to request creating a new instance of meta-interpreters; and the createdInterpreter message is sent back from the object with a meta-interpreter identifier created if successful. We can communicate with the meta-interpreter instance using this identifier.
- The insertModule and insertView messages are sent to a meta-interpreter instance to request loading a module and a view into the meta-interpreter, respectively; the insertedModule and insertedView messages are sent back from the meta-interpreter if the module and the view are loaded successfully, respectively.
- The reduceTerm message is sent to a meta-interpreter instance to request simplifying (or reducing) a term under a module loaded into the meta-interpreter before; and the reducedTerm message is sent back from the meta-interpreter along with the result of the simplification when it is complete.
- The quit message is sent to a meta-interpreter instance to request stopping the meta-interpreter; and the bye message is sent back as soon as the meta-interpreter is closed successfully.

Let us specify a system that has a server and the server is requested to increment a natural number. However, the increment of the natural number is not carried out by the server but by a meta-interpreter independently. For the sake of simplicity, we ignore handling errors. The system can be specified by the following system module:

```
mod SERVER is
  extending META-INTERPRETER .
--
  op Server : -> Cid .
  op aServer : -> Oid .
  op val :_ : Nat -> Attribute .
--
  vars O O' MI : Oid .
  var AS : AttributeSet .
  var T : Term .
  var RT : Type .
  vars N C : Nat .
  rl [loadMod]: < O : Server | AS >
    createdInterpreter(O, O', MI)
  => < O : Server | AS >
----
    insertModule(MI, O, upModule('NAT, true)) .
--
    crl [redTerm]: < O : Server | val : N, AS >
    insertedModule(O, MI)
  => < O : Server | val : N, AS >
    reduceTerm(MI, O, 'NAT, T)
  if T := '_+_[upTerm(N), upTerm(1)] .
  rl [quit]: < O : Server | val : N, AS >
    reducedTerm(O, MI, C, T, RT)
--
    => < O : Server | val : downTerm(T, 0), AS >
    quit(MI, O) .
  rl [bye]: < O : Server | AS > bye(O, MI)
  => < O : Server | AS > .
endm
```

where the four rewrite rules describe how the system interacts with meta-interpreters in regards to some messages exchanged to increment the natural number stored in the val attribute of a server. Modules and terms are first meta-presented by means of the upTerm function [32] and then sent to meta-interpreters as in the rewrite rules loadMod and redTerm. Meanwhile, terms returned by meta-interpreters are meta-representations that should be converted into object representations by using the downTerm function before being used as in the rewrite rule quit.

For example, we can request the system to increment a natural number, namely 3, stored in the val attribute of a server and get its result by the following command:

```
Maude> erewrite in SERVER :
  <> < aServer : Server | val : 3 >
--
    createInterpreter(interpreterManager, aServer, no
result Configuration:
  <> < aServer : Server | val : 4 >
```

where the <> symbol means communicating with external objects, namely meta-interpreters, aServer is an object identifier of the class Server, and none is an empty set of options. Rewriting with external objects is conducted by the external rewrite command **erewrite** in Maude.

3 MAUDE-NPA

Maude-NPA [9] is a verification tool for analyzing cryptographic protocols modulo an equational theory, which is written in Maude and so its specifications follow the Maude syntax. This section describes Maude-NPA in a nutshell (see [9] for more details).

3.1 A protocol specification in Maude-NPA

A protocol specification specified in Maude-NPA consists of three modules as follows:

- PROTOCOL-EXAMPLE-SYMBOLS defines the syntax of the protocol ~~, which consists of~~ as sorts, subsorts, and operators.
- PROTOCOL-EXAMPLE-ALGEBRAIC defines the algebraic properties of the operators ~~, consisting of~~ as equational rules (equations) and equational axioms (axioms).
- PROTOCOL-SPECIFICATION defines the actual behaviors of the protocol using the Dolev-Yao strand space model [23], [24] and the attack states, consisting of at least three equations as follows:
  - STRANDS-DOLEVYAO describes the capabilities of the intruder.
  - STRANDS-PROTOCOL describes the strands of the honest principals. Note that the intruder can also do ~~anything that~~ what honest principals can do.
  - ATTACK-STATE(N), where N is a natural number representing the attack ID, allows us to specify an attack for which we would like to prove the protocol secure or insecure.

Let us briefly describe how to specify the Needham-Schroeder Public Key (NSPK) protocol [6] in Maude-NPA as an example (see [9] for detail). The NSPK protocol uses the standard Alice-and-Bob notation with three messages exchanged as follows:

1) $A \rightarrow B : pk(B, A; N_A)$
2) $B \rightarrow A : pk(A, N_A; N_B)$
3) $A \rightarrow B : pk(B, N_B)$

where A and B denote Alice and Bob principal identifiers (names), N_A and N_B denote the nonces generated by A and B, $pk(A, ...)$ and $pk(B, ...)$ represent the encrypted data of names and/or nonces with the public keys of A and B, respectively, where ... denotes the names and/or nonces omitted, and $A \rightarrow B$ denotes that the data is sent from A to B.

In the module PROTOCOL-EXAMPLE-SYMBOLS for the NSPK protocol, we use some built-in sorts in Maude-NPA such as the sort Msg that represents the messages in the protocol, the sort Fresh that is used to identify terms that must be unique, and the sort Public that is used to identify terms that are publicly available. Besides, we define some sorts Name, Key, and Nonce to distinguish names, keys, and nonces, respectively. All three sorts are subsorts of the sort Msg. Principal names are publicly available and used as keys to encrypt data, and so the sort Name is a subsort of both sorts Public and Key.

```
--- Sort information
~~sorts Name Nonce Key .~~
~~subsort Name Nonce Key < Msg .~~
~~subsort Name < Key .~~
~~subsort Name < Public .~~
sorts Name Nonce Key . subsort Name Nonce Key < Msg .
subsort Name < Key . subsort Name < Public .
```

The following operators are defined: pk and sk for public and private key encryption, n for nonces that are unguessable values for principals, three constants a (for Alice), b (for Bob) and i (for Intruder), and _;_ for message concatenation.

```
--- Encoding operators for public/private encryption
op pk : Key Msg -> Msg [frozen] .
op sk : Key Msg -> Msg [frozen] .
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
--- Message concatenation
op _;_ : Msg Msg -> Msg [gather (e E) frozen] .
```

~~where the~~ The frozen attribute tells Maude not to attempt to apply rewrites at arguments of those symbols and ~~the~~ it is required by Maude-NPA (see Section 3.2 in the Maude-NPA manual [9] for more details). The gather (e E) attribute indicates that the operator _;_ should be parsed as right-associativity.

In the module PROTOCOL-EXAMPLE-ALGEBRAIC for the NSPK protocol, we define two cryptographic equations (primitives) describing the relationship between the public and private key encryption, which is called encryption/decryption cancellation.

```
~~var Z : Msg .~~
~~var K : Key .~~
var Z : Msg . var K : Key .
--- Encryption/decryption cancellation
eq pk(K,sk(K,Z)) = Z [variant] .
eq sk(K,pk(K,Z)) = Z [variant] .
```

In the module PROTOCOL-SPECIFICATION for the NSPK protocol, we first specify the protocol itself and the intruder capabilities using strands in which each strand is a sequence of positive and negative messages describing each principal's executing a protocol or the intruder's performing actions as follows:

$$:: r_1, \ldots, r_j :: [\, m_1^{\pm}, \ldots, m_i^{\pm} \mid m_{i+1}^{\pm}, \ldots, m_k^{\pm}]$$

where $r_1, \ldots, r_j$ are variables of the sort Fresh uniquely generated in the strand, a positive message m^+ describes sending the message m, and a negative message m^- describes receiving the message m. The vertical bar is used to distinguish between present and future when the strand appears in a state. Messages before the bar were sent or received in the past, while messages after the bar will be sent or received in the future. Strands are also used in a protocol specification to build rewrite rules for the backward narrowing and so the vertical bar in such strands is not significant. By convention, each strand of a protocol specification is assumed to have a vertical bar immediately after nil (denoting empty), indicating no messages have been exchanged at the beginning.

As a starting point, we declare all variables with their sorts used in the module for the NSPK protocol as follows:

```
var K : Key .
vars X Y Z : Msg .
vars r r' : Fresh .
vars A B : Name .
var K : Key .          vars X Y Z : Msg .
vars r r' : Fresh .    vars A B : Name .
vars N N1 N2 : Nonce .
```

For specifying the intruder capabilities, all intruder strands follow a convention: a possible sequence of negative variables followed by at least one positive message combining previous variables under a function symbol as follows:

```
eq STRANDS-DOLEVYAO
  = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
    :: nil :: [ nil | -(X ; Y), +(X), nil ] &
    :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
    :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
    :: nil :: [ nil | -(X), +(pk(K,X)), nil ] &
    :: nil :: [ nil | +(A), nil ]
  [nonexec] .
```

where the The `nonexec` attribute denotes that the equation is not executable by rewriting but allowed for narrowing, and it is required by Maude-NPA (see Section 3.4 in the Maude-NPA manual [9] for more details). We recall that `i` is a constant for Intruder. For example, the intruder can concatenate two arbitrary messages to introduce a new message to the network as described in the first strand of the intruder capabilities.

For specifying the behavior of the honest protocol principals in the NSPK protocol, we represent each role of an initiator (Alice) and a responder (Bob) as a strand containing received messages and sent messages as follows:

```
eq STRANDS-PROTOCOL
   = :: r :: [ nil | +(pk(B,A ; n(A,r))),
       -(pk(A,n(A,r) ; N)), +(pk(B, N)), nil ] &
     :: r :: [ nil | -(pk(B,A ; N)),
       +(pk(A, N ; n(B,r))), -(pk(B,n(B,r))), nil ]
  [nonexec] .
```

Attack states describe not only single concrete attacks but also attack patterns where variables are used in terms. We can specify many attack states in which each attack state is designated with a unique natural number as the attack ID. For each attack state specified, Maude-NPA builds an actual attack state at the beginning to conduct the backward analysis that consists of five components separated by the symbol `||` in the following order:

1) the set of current strands that shows how advanced each strand in the execution process by the position of the bar symbol in the strand,
2) the current intruder knowledge that represents what messages the intruder knows (symbol `_inI`) or does not know yet (symbol `_!inI`),
3) the sequences of actual messages exchanged so far from the attack state,
4) the ghost list that is auxiliary information for optimization in the super lazy intruder technique to reduce the state space [9], and
5) the never pattern that is used for authentication attacks.

When specifying an attack state, we should specify only the first two components: (i) a set of strands expected to appear in the attack state, and (ii) some positive intruder knowledge. For the NSPK protocol, we specify an attack state whose attack ID is 0 as follows:

```
eq ATTACK-STATE(0)
   = :: r :: [ nil, -(pk(b,a ; N)),
     +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
     || n(b,r) inI, empty
     || nil
     || nil
     || nil
     || n(b,r) inI, empty || nil || nil || nil
   [nonexec] .
```

where the attack requires that when Bob finishes the protocol, the intruder must learn the nonce generated by Bob. In order to describe the nonce `n(b,r)` leaked to the intruder, we include Bob's strand in the attack. To conduct the attack, we can use the following command in Maude-NPA:

```
Maude> reduce in MAUDE-NPA : run(0, unbounded) .
reduce in MAUDE-NPA : run(0, unbounded) .
```

where 0 is the attack ID specified and `unbounded` is the unbounded depth used in the attack. The result of the command returns an initial state as a counterexample (see [9] for detail). It means that the NSPK protocol does not satisfy the nonce secrecy property (NSP) as the flaw found by Lowe [5] with a man-in-the-middle attack.

3.2 How Maude-NPA works

Maude-NPA starts from an attack state, a final insecure state, to perform a backward reachability analysis that determines whether or not it is reachable from an initial state, which has no further backward steps. If that is the case, the initial state is a counterexample. The backward search is performed by backward narrowing with symbolic execution because the attack state is a term with logical variables. Each backward narrowing step can be regarded as the reverse direction of a state transition, such as sending or receiving a message by principals, or manipulating a message by intruders. Given a symbolic state, a backward narrowing step is performed to return a previous symbolic state in the protocol. Thereby, we can obtain all successor states (in the backward sense) from the state by performing the backward narrowing just by one step.

We can divide the whole process of Maude-NPA into two main stages. In the first stage, given a protocol specification $\mathcal{P}$ and an equational theory $E_{\mathcal{P}}$, Maude-NPA needs to do as follows:

- Extracting the attack state St from the protocol given an attack ID.
- Building rewrite rules $R_{\mathcal{P}}$ based on the behavior of the protocol specified in the form of intruder and regular strands along with some pre-defined rewrite rules in the Maude-NPA specification.
- Generating grammars that represent infinite sets of states unreachable for the intruder to reduce the state space.

In the second stage, Maude-NPA performs the backward narrowing reachability analysis from the attack state St

using the relation $\rightsquigarrow_{R_{\mathcal{P}}^{-1},E_{\mathcal{P}}}$ where $R_{\mathcal{P}}^{-1}$ is the set of rewrite rules derived from $R_{\mathcal{P}}$ by inverting its rewrite rules. Maude-NPA basically uses a breadth-first search to explore the state space. There are three main steps needed to do for each layer exploration as follows:

- The first step is to generate all successor states for the next layer given a set of states in the current layer. The initial layer is layer 0, the attack state is located at layer 0, and all successor states of the attack state with respect to the relation $\rightsquigarrow_{R_{\mathcal{P}}^{-1},E_{\mathcal{P}}}$ are located at layer 1. This step also consists of almost all techniques to reduce the state space except for the transition subsumption technique, which is used in the second step below.
- The second step is to simplify the successor states by the transition subsumption technique for removing states that are subsumed by either other states in the successor states or visited states (history states).
- Lastly, the third step filters the states from the previous step using the history states to avoid state duplications and rules out initial states as counterexamples. The remaining states without duplication are then stored in the history states as visited states and the depth bound is decreased by one.

The cycle repeats until any initial states (counterexamples) are found, a depth bound is reached, or no states are found for the next layer.

The first step in the second stage actually performs the backward narrowing just by one step to obtain all successor states from a given set of states in a layer. The successor states then go through a series of optimization steps, such as giving priority to input messages in strands, early detection of inconsistent states, the super lazy intruder, and filtering states by the grammars. Given a set of states in layer l, for each state in the set, Maude-NPA performs the backward narrowing to obtain its successor states in layer $l + 1$, which is referred to as step (1) in this paper. The backward narrowing for each given state from the set of states can be executed independently, which opens an opportunity for parallelization. In the next section, we will describe how to parallelize step (1) at each layer in which the successor states are generated in parallel. Note that the parallel version does not alter the number or form of the states in the state space.

The second step in the second stage plays an important role to reduce the state space in Maude-NPA, which is referred to as step (2), which may transform an infinite-state system into a finite one [33] and is also time-consuming because of its complexity. Basically, it performs two sub-steps for the transition subsumption as follows:

- First, for each state in the successor states obtained from step (1), we check whether the state is implied by other states in the successor states, which is referred to as step (2.1) in this paper. If so, the state is discarded. Otherwise, we keep the state. Once step (2.1) is complete, we obtain a set of states such that no state is implied by other states.
- Second, for each state in the states obtained from step (2.1), we check whether the state is implied by some states in the history states, which is referred to as step (2.2) in this paper. If so, the state is discarded. Otherwise, we keep the state. Once step (2.2) is complete, we obtain a set of states such that no state is implied by any state in the history states. The set of states is then used as the input for the third step of the second stage, which is referred to as step (3).

TABLE 1: The number of states after step (1), step (2), and step (3) at each layer for the NSPK protocol

Layers	#inputStates	#states after completing			#initStates
		step (1)	step (2)	step (3)	
layer 1	1	5	4	4	0
layer 2	4	9	6	6	0
layer 3	6	9	4	4	0
layer 4	4	5	2	2	0
layer 5	1	2	1	1	0
layer 6	1	2	2	2	0
layer 7	2	4	4	4	1

The transition subsumption is a complex and heavy task in Maude-NPA when the number of successor states and history states is considerably large. Therefore, we can improve the transition subsumption in Maude-NPA by parallelizing step (2.1) and step (2.2) at each layer, which are described in detail in the next section. Meanwhile, we do not conduct step (3) in parallel because the task in step (3) is not complex and can be completed quickly in sequence even when the number of history states is considerably large.

For the NSPK protocol with the attack ID 0 formalized above, we use Maude-NPA to analyze the number of successor states after completing step (1), step (2), and step (3) at each layer up to layer 7 (or depth 7), where a counterexample is found by Maude-NPA. The data are shown in Table 1. The first column denotes the name of each layer. The second column denotes the number of input states for each layer. Note that the number of input states for layer 1 is one because of only one attack state concerned, while the number of input states for each of the other layers is the number of states after completing step (3) of its previous layer. The third, fourth, and fifth columns denote the number of states after completing step (1), step (2), and step (3) at each layer, respectively. The last column denotes the number of counterexamples found during analysis. We can see that even though the NSPK protocol is simple, step (2) still plays a crucial role in reducing the number of states at each layer. For example, we can reduce the number of states obtained from step (1) from 9, 9, 5 to 6, 4, 2 at layers 2, 3, and 4 after completing step (2), respectively. That demonstrates the important role of the transition subsumption in step (2), which is parallelized in this present paper.

4 Parallel Maude-NPA and Its ~~Tool~~ Support Tool

Par-Maude-NPA is implemented in Maude to conveniently extend the implementation of what has been developed in Maude-NPA. We use object-based programming that can model an object-based system and meta-interpreters that can handle jobs independently to build a parallel version of Maude-NPA with a master-worker model. For more details about object-based systems and meta-interpreters in Maude, the reader is

referred to our supplementary document publicly available at https://github.com/canhminhdo/parallel-maude-npa/blob/master/supplement/document.pdf.

4.1 How to parallelize Maude-NPA

As mentioned above, we parallelize the backward narrowing and the transition subsumption at each layer in Maude-NPA. In our tool, a master maintains a shared cache that is a set of visited states (history states), while each worker maintains some shared information, such as the module used to conduct the backward narrowing and the grammars generated from the protocol under verification for optimization. The use of the shared cache prevents jobs that have been processed from being assigned to workers. The use of the shared information prevents loading them again from each worker whenever it is requested to handle a job. We maintain the status of the master whose value is one of the following constants: *narrowing*, *implication*, *implicationH*, *filter*, and *stop* to distinguish that the master is processing step (1), step (2.1), step (2.2), step (3), and checking for termination, respectively. Note that those steps are performed in this order at each layer in the tool. We then use four constants: *narrowing*, *implication*, *combination*, and *implicationH* to identify four kinds of jobs that are sent from the master to workers because we use different kinds of jobs to parallelize each step. *narrowing* jobs are used in step (1), *implication* and *combination* jobs are used in step (2.1), and *implicationH* jobs are used in step (2.2). Depending on each kind of jobs, a worker is requested to handle a different task as follows:

The number of states after step (1), step (2), and step (3) at each layer for the NSPK protocol **step (1) step (2) step (3)** layer 1 1 5 4 4 0layer 2 4 9 6 6 0layer 3 6 9 4 4 0layer 4 4 5 2 2 0layer 5 1 2 1 1 0layer 6 1 2 2 2 0layer 7 2 4 4 4 1

- *narrowing* requests a worker to conduct the backward narrowing just by one step for a given state in step (1). The result is a set of successor states reachable from the state.
- *implication* requests a worker to conduct the implication for a given set of states in step (2.1). The result is a set of states such that no state can be implied by other states in the set.
- *combination* requests a worker to combine two given sets of simplified states in step (2.1) by a slightly different implication, which is explained in detail later. Note that each set of states has been simplified by the implication before. The result is a set of states such that no state can be implied by other states in the set.
- *implicationH* requests a worker to conduct the implication for a set of states with history states given in step (2.2). The result is a set of states such that no state in the set can be implied by any state in the history states.

There is only one kind of job that is sent from workers to the master. As soon as a worker completes a job assigned to it by the master, the worker sends its result in the form of a set of states to the master, delivering a job or a bunch of jobs made by the worker to the master. Depending on the status of the master, the jobs are stored accordingly. The very initial job is made by the master, while all the other jobs are made by workers and basically sent to the master. Jobs are assigned to workers by the master unless the jobs have been tackled.

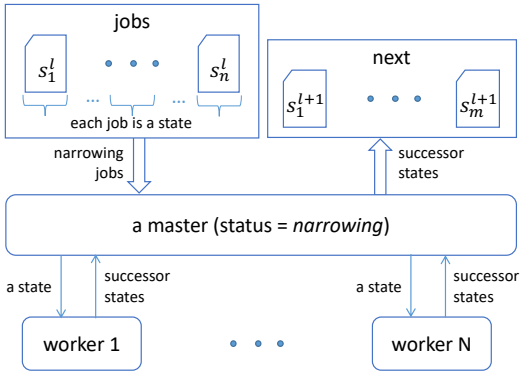


Fig. 1: Conducting the backward narrowing in step (1) from the states at layer l in parallel.

Fig. 1 shows an overview of how to conduct step (1) in parallel for states located at layer l, where the slim arrows denote messages sending and receiving, and the thick arrows denote input jobs and output states at the master side. The status of the master is currently *narrowing*. Given a set of states $s_1^l, \ldots, s_n^l$ stored in the object field `jobs` at layer l as input, for each state, the master constructs a job whose type is *narrowing* job and sends it to a worker. As soon as a When the worker receives the job, it conducts the backward narrowing just by one step from the state encapsulated in the job independently to obtain its successor states at layer $l+1$. The worker then sends the successor states to the master and waits for a new job. The master will store the successor states into the what it is supposed to do. The results from workers will be stored in the object field `next` as output, all possible next jobs (successor states) of the next layer , namely layer for layer $l+1$. If there are neither unprocessed jobs left nor jobs being processed by workers, step (1) is complete. If `next` is not empty, the master changes its status to *implication* to move to conduct step (2.1). Otherwise, the master changes its status to *stop* to terminate subsequently.

Fig. 2 and Fig. 3 show an overview of how to conduct step (2.1) in parallel for those successor states in `next` obtained from step (1). The status of the master is currently *implication*. Firstly, the master The master first divides the successor states in `next` into multiple partitionsin which , where each partition is a set of states (see Fig. 2). The number of partitions is calculated based on the number of workers and the minimum number of states that should be assigned to a worker, such that the number of partitions is smaller than or equal to the number of workers. How to calculate the number of partitions How to conduct the partition will be described in detail later. If the number of partitions is one, we do not conduct it in parallel but in sequence and then change the status of the master to *implication*. For each partition, the master constructs a job whose type is an *implication* job and sends it to a worker.

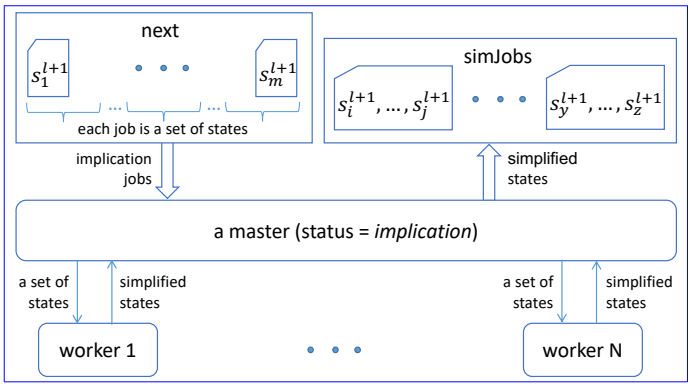


Fig. 2: Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel.

As soon as a When the worker receives the job, it conducts the implication inside the set of states encapsulated in the job to obtain a set of simplified states such that no state is implied by other states in the set. The worker then sends the simplified states to the master that will be does what it is supposed to do. The results from workers are stored in the object field `simJobs`, a queue of simplified jobsin which , where each job is a set of simplified states. If there are no jobs

If no jobs are being processed by workers, the master moves to combine the simplified jobs in `simJobs` (see Fig. 3). For every two simplified jobs in `simJobs`, the master constructs a job whose type is *combination* job and sends it to a worker. As soon as a When the worker receives the job, it combines the two sets of simplified states encapsulated in the job by a slightly different implication, which will be described in detail later. The result is a set of simplified states such that no state is implied by other states in the set. The worker then sends the set of simplified states to the master and waits for a new job. The master will store the set of simplified states into does what it is supposed to do. Each result from a worker is stored in `simJobs` as a simplified job so that it can be combined with another set of simplified states subsequently. If there is later. If only one simplified job is left in `simJobs` and no jobs are being processed by workers, step (2.1) is complete. The master changes its status to *implicationH* to move to conduct step (2.2).

Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel.

Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel (continuously).

Conducting the transition subsumption in step (2.2) for states with history states at layer $l+1$ in parallel.

Fig. 4 shows an overview of how to conduct step (2.2) in parallel from with a set of simplified successor states obtained states from step (2.1). The status of the master is currently *implicationH* and there is only one simplified job is left in `simJobs`that is a set of states. The master dequeues `simJobs` to obtain the set of simplified states and divides it into multiple partitionsin which , where each partition is a set of states. The number of partitions is calculated based on the number of workers, the number of history states, and the minimum number of states that should be assigned to a worker, such that the number of partitions is smaller than or equal to the number of workers. How to calculate the number of partitions How to conduct the partition will be described in detail later. If the number of partitions is one, we do not conduct it in parallel but in sequence and then change the status of the master to *filter*. For each partition with history states, the master constructs a job whose type is an *implicationH* job and sends it to a worker. As soon as a When the worker receives the job, it conducts the implication for the set of states with the history states encapsulated in the job as described in step (2.2). The result is a set of simplified states such that no state in the set is implied by any state in the history states. The worker then sends it to the master to be does what it is supposed to do. The results from workers are stored in `next`. If there are no jobs being processed by workersworkers are processing no jobs, step (2.2) is complete. The master changes its status to *filter* to conduct step (3).

The master then conducts step (3) in sequence and changes its status to *narrowing* for the next layer. The cycle repeats until any initial states (counterexamples) are found, a depth bound is reached, or no states are found for the next layer. If that is the case, then the master changes its status to *stop* to terminate the tool subsequently.

4.2 Job scheduling by the master

In summary, the master is mainly in charge of the following tasks. For step (1), the master is in charge of collecting all successor states (jobs) from workers. For step (2.1), if it is necessary to conduct step (2.1) in parallel, the master is in charge of collecting (simplified) jobs from workers in which each job is a set of simplified states. Otherwise, the master is in charge of conducting step (2.1) in sequence. For step (2.2), if it is necessary to conduct step (2.2) in parallel, the master is in charge of collecting all states (jobs) from workers such that no state is implied by any state in history states. Otherwise, the master is in charge of conducting step (2.2) in sequence. For step (3), the master is in charge of conducting step (3) in sequence as Maude-NPA does. In addition, the master is in charge of distributing (or assigning) collecting jobs and distributing unprocessed jobs to workers. The master can also stop the tool if some conditions happen as described above. The To do so, the master uses a set of states to store jobs (i.e., `jobs` and `next`) `next`) in step (1), a queue of sets of states to store (simplified) simplified jobs (i.e., `simJobs`) in step (2.1), a set of history states (i.e., `history`), and a queue of worker identifiers (i.e., `workers`) to distribute jobs to workers in a well-balanced way. Note that the number of jobs in step (2.2) is always smaller than or equal to the number of available workers in the tool. Hence, we simply distribute each job to a worker on the fly and do not need to store such jobs.

Algorithm 1 shows the pseudocode for job scheduling by the master. `workers`, `jobs`, `next`, `simJobs`, `history`, and `status` are used for what are described in the previous subsection. Besides, we use the object field `initStates` to store initial states (counterexamples) found by the tool. The first stage in Maude-NPA is conducted at line 1 using the `initialize` function and then we can obtain a module `M` representing the relation $\rightsquigarrow_{R_{\mathcal{P}}^{-1},E_{\mathcal{P}}}$, gram-

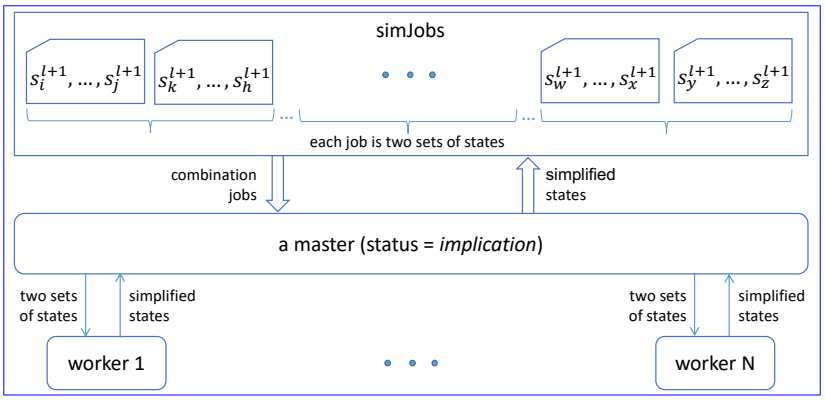


Fig. 3: Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel (continuously).

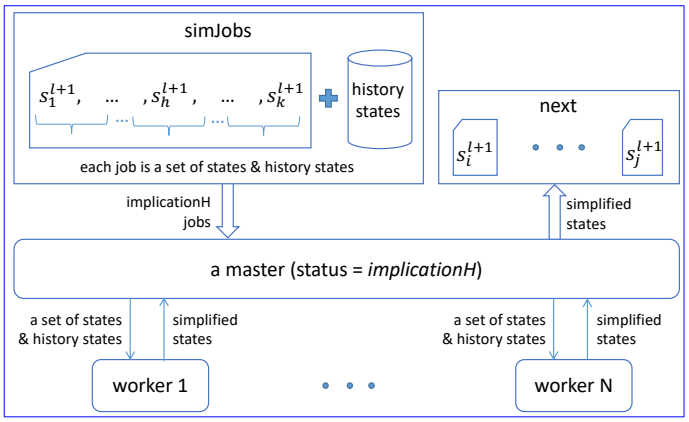


Fig. 4: Conducting the transition subsumption in step (2.2) for states with history states at layer $l+1$ in parallel.

mars `GS`, and an attack state `IS` from a protocol specification $\mathcal{P}$. We then prepare N workers at line 2 using the `initializeInterpreters` function. Note that each worker represents a meta-interpreter in which some shared information (e.g., `M` and `GS`) and many other modules from Maude and Maude-NPA are loaded so as to handle the jobs assigned to workers by the master. The code fragment at lines 3 – 5 is used for initialization. The `while` loop in the code fragment at lines 6 – 13 conducts the backward narrowing analysis for the protocol under verification. The master calls the `narrowingStep`, `impliationStep`, `implicationWithHistoryStep`, and `filterStep` functions in order to conduct step (1), step (2.1), and step (2.2) in parallel, and step (3) in this order at each layer, respectively. At the end of the `while` loop, the master checks whether `status` is *stop* for termination. If so, the master closes all connections and returns `initStates` as the verification result. Otherwise, the master keeps on doing for the next layer. The code fragment at lines 14 – 22 defines the `receivingData` function that is used in the `narrowingStep`, `impliationStep`, and `implicationWithHistoryStep` functions to receive data from workers. Whenever the master receives `DATA` from each $worker_k$, the worker identifier $worker_k$ is first enqueued to `workers` and the set of states `IST` deconstructed from `DATA` is stored accordingly to either `next` or `simJobs` based on with respect to the current status of the master.

Algorithm 1: Job scheduling by the master.

input : $\mathcal{P}$ – a protocol specification
Id – an attack ID in the protocol specification
$BStep$ – the maximum number of backward steps
N – a number of workers
output: *empty or counterexamples*

```
1  (M, GS, IS) ← initialize(P, Id, BStep);
2  workers ← initializeInterpreters(M, GS, N);
3  jobs ← {IS}; history ← {IS};
4  (next, simJobs, initStates) ← (empty, empty, empty);
5  status ← narrowing;
6  while True do
       /* conduct step (1) in parallel    */
7      narrowingStep();
       /* conduct step (2.1) in parallel  */
8      implicationStep();
       /* conduct step (2.2) in parallel  */
9      implicationWithHistoryStep();
       /* conduct step (3) in sequence    */
10     filterStep();
       /* check for termination           */
11     if status = stop then
12         closeAllConnections();
13         return initStates;
14 function receivingData() is
15     for k ← 1 to N do
16         if DATA ← recv(worker_k) then
17             enqueue(workers, worker_k);
18             (IST) ← DATA;
19             if status = narrowing or
                 status = implicationH then
20                 next ← next ∪ IST;
21             if status = implication then
22                 enqueue(simJobs, {IST});
```

Algorithm 2 shows the pseudocode of the `narrowingStep` function in which we conduct the backward narrowing in step (1) in parallel for a given set of states stored in `jobs` at the current layer being concerned

Algorithm 2: Conducting the backward narrowing in step (1) at each layer in parallel.

```
1  function narrowingStep() is
2      if status != narrowing then
3          return;
4      while True do
5          receivingData();
6          while not isEmpty(workers) and
            not isEmpty(jobs) do
7              worker ← dequeue(workers);
8              IS ← getOne(jobs);
9              send(worker, narrowing, IS);
10         if isEmpty(jobs) and size(workers) = N
            then
11             if not isEmpty(next) then
12                 status ← implication;
13             else
14                 status ← stop;
15             return;
```

Algorithm 3: Conducting the implication in step (2.1) for states at each layer in parallel.

```
1  function implicationStep() is
2      if status != implication then
3          return;
4      nW ←
        neededWorkersForSim(size(next), N, simBatch);
5      if nW = 1 then
           /* step (2.1) in sequence        */
6          next ← simplifyByImplicationL(next);
7          status ← implicationH;
8          return;
       /* step (2.1) in parallel            */
9      (unusedW, usedW) ← getWorkers(workers, nW);
10     workers ← unusedW;
11     sendJobs(usedW, implication, next);
12     next ← empty;
13     while True do
14         receivingData();
15         while not isEmpty(workers) and
            size(simJobs) > 1 do
16             worker ← dequeue(workers);
17             IST1 ← dequeue(simJobs);
18             IST2 ← dequeue(simJobs);
19             send(worker, combination, IST1, IST2);
20         if size(simJobs) = 1 and size(workers) = N
            then
21             status ← implicationH;
22             next ← dequeue(simJobs);
23             return;
```

(see Fig. 1 as well). The code fragment at lines 2 – 3 ensures that `status` is *narrowing* before processing further. The `receivingData` function is called to collect jobs sent from workers at line 5. The code fragment at lines 6 – 9 shows how to distribute jobs to workers. The code fragment at lines 10 – 15 checks whether step (1) is complete. If so, the master changes its status accordingly as described ~~with~~ in Fig. 1 and moves to step (2.1). ~~At this point, we have completed step (1) in parallel and the successor states are currently stored in~~ `next`~~.~~

Algorithm 3 shows the pseudocode of the `implicationStep` function in which we conduct the implication in step (2.1) in parallel for the successor states ~~obtained~~ in `next` from step (1) (see Fig. 2 and Fig. 3 as well). The ~~master first~~ code fragment at lines 2 – 3 ensures that `status` is *implication* before processing further~~in the code fragment at lines 2 – 3.~~. It then calculates the number of partitions ~~at~~ in step (2.1) ~~in the form~~ by means of the number of needed workers at line 4. The size of each partition may affect the running performance of the tool. Because if each partition contains a small number of states (a light job), a worker may finish its task quickly and so the communication cost may be larger than the benefit able to be gained from parallelization. Hence, we use ~~the parameter~~a parameter, called `simBatch`, that denotes the minimum number of states in each partition that should be assigned to a worker. ~~Based on~~ We calculate the number of needed workers as follows:

$$min\{\lceil C \div \texttt{simBatch} \rceil, N\}$$

where C and N are the number of successor states in `next`, ~~the number of workers N, and~~ `simBatch`~~, we calculate~~ and the number of ~~needed workersfor parallelization as follows:~~ workers, respectively. This guarantees that each selected worker ~~has~~ will have a job such that the number of states is at least `simBatch` states. Note that `simBatch` is set to 20 as default and the number of needed workers is a positive natural number ~~and~~ less than or equal to N ~~because we only have N workers available~~in the formula. If the number of needed workers is one, the master conducts step (2.1) in sequence in the code fragment at lines 6 – 8. Otherwise, it conducts step (2.1) in parallel in the code fragment at lines 9 – 23 as described ~~with~~ in Fig. 2 and Fig. 3. In ~~both cases~~either case, the status of the master is set to *implicationH* at the end to move to step (2.2)~~and the states after step (2.1) are currently stored in~~ `next`~~.~~. `next`.

Algorithm 4 shows the pseudocode of the `implicationWithHistoryStep` function in which we conduct the implication with history states in step (2.2) in parallel for the states ~~obtained~~ in `next` from step (2.1) (see Fig. 4 as well). The ~~master first~~ code fragment at lines 2 – 3 ensures that `status` is *implicationH* before processing further~~in the code fragment at lines 2 – 3.~~. It then calculates the number of partitions at step (2.2) ~~in the form~~ by means of the number of needed workers at line 4. Note that for simplicity, in Fig. 4 we divide the set of successor states in the last job of `simJobs` on the fly instead of assigning them to `next` and then dividing them from `next` as what we have been doing so far. The size of each partition and the number of history states may affect the running performance of the tool because of the same reason as described in step (2.1). Hence, we ~~use the parameter~~also use a parameter, called `simBatchH`, that denotes the minimum number of the multiplication of the number of history states and the number of states in each

Algorithm 4: Conducting the implication in step (2.2) for states with history states at each layer in parallel.

```
1  function implicationWithHistoryStep() is
2      if status != implicationH then
3          return;
4      nW ← neededWorkersForSimH(size(history),
         size(next), N, simBatchH);
5      if nW = 1 then
           /* step (2.2) in sequence        */
6          next ← simplifyByImplicationH(history, next);
7          status ← filter;
8          return;
       /* step (2.2) in parallel            */
9      (unusedW, usedW) ← getWorkers(workers, nW);
10     workers ← unusedW;
11     sendJobs(usedW, implicationH, history, next);
12     next ← empty;
13     while True do
14         receivingData();
15         if size(workers) = N then
16             status ← filter;
17             return;
```

partition that should be assigned to a worker. ~~Based on~~ We calculate the number of needed workers as:

$$min\{\lceil C \times H \div \texttt{simBatchH} \rceil, N\}$$

where C, H, and N are the number of states in `next`, the number of history states in `history`, ~~the number of workers N, and~~ `simBatchH`~~, we calculate~~ and the number of ~~needed workersto conduct step (2.2) in parallel as follows:~~ workers, respectively. This guarantees that each selected worker ~~has~~ will have a job such that the multiplication of the number of states in its partition and the number of history states is at least `simBatchH`. Note that `simBatchH` is set to 50 as default. If the number of needed workers is one, the master conducts step (2.2) in sequence in the code fragment at lines 5 – 8. Otherwise, it conducts step (2.2) in parallel in the code fragment at lines 9 – 17 as described with Fig. 4. Note that as soon as jobs are distributed to workers, `next` is set to empty to store the results from workers subsequently at line 12. In ~~both cases~~either case, the status of the master is set to *filter* at the end to move to step (3)~~and the states after step (2.2) are currently stored in `next`~~. `next`

Algorithm 5 shows the pseudocode of the `filterStep` function in which the master conducts step (3) in sequence. It first ensures that `status` is *filter* before processing further in the code fragment at lines 2 – 3. The master then filters states by the history states, rules out initial states, updates `jobs`, `next`, `BStep` and `history`, and sets `status` to *narrowing* for the next layer in the code fragment at lines 4 – 7. At the end, the master checks for termination in the code fragment at lines 8 – 10. At this point, we have completed step (3) and move to check whether the status of the master is *stop* for termination in the code fragment at lines 11 – 13 in Algorithm 1.

Algorithm 5: Filtering state duplications and ruling out initial states at each layer in step (3) in sequence.

```
1  function filterStep() is
2      if status != filter then
3          return;
4      (INIT, IST) ←
         filterWithHistoryAndInit(M, history, next);
5      (jobs, next, BStep) ← (IST, empty, BStep − 1);
6      history ← history ∪ IST;
7      status ← narrowing;
8      if not isEmpty(INIT) or BStep = 0 or
         isEmpty(jobs) then
9          initStates ← INIT;
10         status ← stop;
```

Algorithm 6: Job handling by workers.

```
input : M – the module used to conduct the
        backward narrowing
        GS – the grammars generated from the
protocol under verification
1  while isOpen() do
2      if DATA ← recv(master) then
3          if (narrowing, IS) ← DATA) then
4              IST ← nextBackNarrowForParallel(M, GS, IS);
5              send(master, IST);
6          else if (implication, IST) ← DATA then
7              IST′ ← simplifyByImplicationL(IST);
               send(master, IST′);
8          else if (combination, IST1, IST2) ← DATA
             then
9              IST ← combineSimplifyByImplicationL(
               IST1, IST2);      send(master, IST);
10         else if
             (implicationH, History, IST) ← DATA then
11             IST′ ← simplifyByImplicationH(History, IST);
12             send(master, IST′);
```

4.3 Job handing by workers

Algorithm 6 shows the pseudocode for workers handling the jobs assigned to them by the master. The code fragment at lines 3 – 14 describes how a worker handles a job based on the type of the job. For example, in the case of *combination* in the code fragment at lines 9 – 11, the worker first deconstructs `DATA` to obtain the two sets of simplified states `IST1` and `IST2`. ~~Note that each set of states has been simplified before.~~ Given `IST1` and `IST2`~~, `IST2`~~, the `combineSimplifyByImplicationL` function combines the two sets in step (2.1) with a slightly different implication. ~~Basically, we do not need to check the implication for the states in each set.~~ For each state in the set `IST1`, if the state is implied by a state in the set `IST2`, then the state is removed from the set `IST1`. For each state in the set `IST2`, if the state is implied by a state in the set `IST1` that has been checked, then the state is removed from the set `IST2`. Thereby, we can obtain a set of ~~states such that no state is implied by other states in the set. The result of the function is then sent to the master as a simplified job. Note that workers terminate~~

~~if and only if the master closes all connections.~~ simplified states, which will be sent to the master as a simplified job. The different procedure between the slightly different implication and the normal implication in Maude-NPA is as follows:

- We do not need to check the implication for states in each set of `IST1` and `IST2` because they have been simplified before as given inputs.
- We combine the two sets of simplified states to obtain a set of simplified states, while the normal simplification only simplifies a set of states.

Note that the simplification process between the two states is used in the same way.

The backward narrowing and simplification functions are complex and tailored for the sequential version of Maude-NPA. Therefore, we intentionally use them as core functions in the parallel version of Maude-NPA without changing the theoretical algorithms behind Maude-NPA. The `nextBackNarrowForParallel` and `combineSimplifyByImplicationL` functions are built based on ~~existing~~ these functions in Maude-NPA, while the `simplifyByImplicationL` `simplifyByImplicationL` and `simplifyByImplicationH` functions are ~~defined~~ existing functions in Maude-NPA to conduct the simplifications for step (2.1) and step (2.2) in sequence, respectively.

~~Job handling by workers.~~
~~$IST \leftarrow nextBackNarrowForParallel(M, GS, IS)$;send(master, IST)~~
~~$IST' \leftarrow simplifyByImplicationL(IST)$~~
~~send(master, IST') $IST \leftarrow combineSimplifyByImplicationL($~~
~~IST1, IST2) send(master, IST)~~
~~$IST' \leftarrow simplifyByImplicationH(History, IST)$;send(master, IST')~~

5 Experiments

5.1 Experiment setup

We have used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory of RAM to conduct experiments. We use Maude-NPA and multiple parallel versions of Maude-NPA including Par-Maude-NPA-1 and Par-Maude-NPA-2 in our case studies. The tool and the case studies for the experiments are publicly available at the webpage https://github.com/canhminhdo/parallel-maude-npa. Besides, the original source code of the case studies and more protocols are listed at http://personales.upv.es/sanesro/Maude-NPA_Protocols/index.html.

5.2 Performance of Par-Maude-NPA-2

We have conducted experiments on various kinds of protocols to confirm the usefulness of Par-Maude-NPA-2 such as symmetric key protocols, homomorphism protocols, exclusive or protocols, API protocols, PKCS protocols, choice protocols, and distance-bounding protocols. The experimental data are shown in Tables~~A–B at~~ 2–3. The first and second columns denote the name of the protocols and the attack ID used in protocol specifications. To see each attack in detail, readers can refer to the protocol specifications publicly available at the two webpages above or in some papers [34], [35], [36], [37], [38], [39]. The third column denotes the result of each analysis that terminates at a depth, where the cross and check symbols denote insecure and secure protocols from attacks, respectively. For insecure protocols, counterexamples are found at the depth, while for secure protocols except for TLS attack protocol with the question mark symbol in the third column of Table~~B~~ 3, no counterexamples are found and no successor states in the next layer from the depth. Because the state space of the TLS attack protocol is huge, we use a bounded depth to conduct an analysis, meaning that the TLS attack protocol is only secure up to the bounded depth in the analysis. Note that other protocols are analyzed at an unbounded depth. The fourth and fifth columns denote the verification time excluding the time taken to generate the grammars for protocols when conducting formal verification with Maude-NPA and Par-Maude-NPA-2, respectively. Grammar generation is not taken into account for execution time because they are not parallelized. The winning tool is indicated by a bold value in either the fourth or fifth column in a row. The sixth column denotes the percentage of improvement when using Par-Maude-NPA-2. If the value is a positive number, namely X, it means that the Par-Maude-NPA-2 is X% faster than Maude-NPA. Conversely, if the value is a negative number, namely $-X$, it means that Maude-NPA is X% faster than Par-Maude-NPA-2. The last column denotes the average number of states at each layer for each worker to handle, respectively. Formal verification experiments terminate as soon as initial states (counterexamples) are found, the depth bound is reached, or no states are found for the next layer.

Our previous tool, Par-Maude-NPA-1, relies on a parallel version of Maude-NPA [1] that uses Maude sockets to communicate the master and workers and so we can flexibly choose to use a shared-memory machine or a distributed environment. Meanwhile, the tool in the present paper, Par-Maude-NPA-2, relies on a new parallel version of Maude-NPA that uses meta-interpreters and so we can use only a shared-memory machine. For the experiments in Tables~~A–B mentioned above~~ 2–3, we use a master and eight workers with a shared-memory machine, the MacPro computer. Notice that we use the same computer and configuration to conduct the experiments with Par-Maude-NPA-2 as what we have done with Par-Maude-NPA-1 in [1]. We classify case studies into simple and complex case studies based on the number of states in their reachable state spaces as well as their verification time. The experimental data say that for simple case studies (25 experiments) whose verification time is less than 40 seconds, Maude-NPA is faster than Par-Maude-NPA-2 because the number of states located at each layer is very small and the verification time is so short that the communication cost between the master and workers as well as the cost to prepare workers and load many modules into each worker become burdensome. However, Par-Maude-NPA-2 still can finish in a reasonably short amount of time. As described above, we use `simBatch` and `simBatchH` to decide whether we should conduct step (2.1) and step (2.2) in parallel, respectively. For simple case studies, we do not need to conduct step (2.1) and step (2.2) in parallel~~because the number of successor states and~~

history states at each layer is small. . For complex case studies (34 experiments) in which the number of states located at each layer is larger, Par-Maude-NPA-2 has a very good performance that is 44% faster than Maude-NPA on average, demonstrating its potential. Among the complex case studies, there are some case studies for which we may not need to conduct step (2.1) and step (2.2) in parallel at some layers, where the number of states and history states at each layer is small, because of the use of `simBatch` and `simBatchH`. For the three protocols Amended Needham Schroeder, YubiKey, and TLS Attack whose verification time is the largest among the protocols used for experiments, Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA by 57%, 69%, and 64%, respectively. Meanwhile, Par-Maude-NPA-1 can only improve the running performance of Maude-NPA by 39%, 30%, and 20%, respectively [1], which demonstrates that Par-Maude-NPA-2 can largely improve running performances of Par-Maude-NPA-1 as well as Maude-NPA. This is because the tool uses meta-interpreters instead of Maude sockets and parallelizes not only the backward narrowing in step (1) but also the transition subsumption in step (2.1) and step (2.2) in Maude-NPA. For step (1), the average number of states at each layer for a worker is measured to let us know how busy each worker is, which reflects the number of states located at each layer. The busier workers are and the deeper the depth bound is, the more benefit we may gain from the use of parallelization for step (1). For step (2.1) and step (2.2), the more successor states and history states at each layer, the more benefit we may gain from the use of parallelizationalso regardless of the improvement in parallelizing step (1). .

In addition, we would like

5.3 Effectiveness of the use of meta-interpreters and the parallelization of the transition subsumption

In order to demonstrate the effectiveness of the use of meta-interpreters and the parallelization of the transition subsumption in step (2.1) and step (2.2). We , we use another version of Par-Maude-NPA-2, called Par-Maude-NPA-1*, in which only the backward narrowing in step (1) is parallelized while the transition subsumption in step (2.1) and step (2.2) are performed in sequence by setting `simBatch` and `simBatchH` to unbounded. Par-Maude-NPA-1* can be regarded as another version of Par-Maude-NPA-1 in which meta-interpreters are used instead of Maude sockets. We conduct experiments for the three protocols whose verification time is the largest among the protocols used for experiments because the number of states at each layer of the three protocols is large and so we can see the difference between Par-Maude-NPA-1, Par-Maude-NPA-1*, and Par-Maude-NPA-2 in running performances. We use one master and eight workers with the MacPro machine to conduct experiments. The experimental data are shown in Table 4. Note that Amended NS denotes the Amended Needham Schroeder protocol. The sixth, eighth, and tenth columns denote the percentage of improvement for Par-Maude-NPA-1Par-Maude-NPA-1, Par-Maude-NPA-1*, and Par-Maude-NPA-2 compared to Maude-NPA, respectively. For the three protocols, Par-Maude-NPA-1* can improve the running performance of Maude-NPA from 39% to 43% (4% increased), 30% to 37% (7% increased), 20% to 50% (30% increased) compared to Par-Maude-NPA-1, respectively, demonstrating that the use of meta-interpreters instead of Maude sockets is efficient. Meanwhile, Par-Maude-NPA-2 can improve the running performance of Maude-NPA from 43% to 57% (14% increased), 37% to 69% (32% increased), 50% to 64% (14% increased) for the three protocols compared to Par-Maude-NPA-1*, respectively, demonstrating that the parallelization of the transition subsumption in step (2.1) and step (2.2) is effective. Therefore, Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA for the three protocols because of the use of meta-interpreters and parallelization of step (1), step (2.1), and step (2.2).

Moreover, we

5.4 Percentage improvement and memory usage

We would like to demonstrate the power of measure the percentage of improvement of Par-Maude-NPA-2 further by conducting more with respect to memory usage by conducting experiments with various numbers of workers for the three protocols whose verification time is the largest among the protocols used for experiments. The experimental data are shown in Table 5. The fifth column denotes the number of workers used in the experiments. The sixth and eighth columns denote the verification time for Par-Maude-NPA-1 and Par-Maude-NPA-2, respectively. The seventh and ninth columns denote the percentage of improvement for Par-Maude-NPA-1 and Par-Maude-NPA-2 compared to Maude-NPA, respectively. We The percentage of improvement and the memory usage are plotted in Fig 5 and Fig 6, respectively.

Regarding the percentage of improvement, we can see that Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA from 39% to 57% (18% increased), 30% to 69% (39% increased), and 20% to 64% (44% increased) for the three protocols compared to Par-Maude-NPA-1, respectively, when eight workers are used. When we increase the number of workers to 16 and 24, the tool still improves the running performance of Maude-NPA further, such as 61%, 76%, and 72% for the three protocols, respectively, when the number of workers is 24. We can see that the but its increment is very slow (see Fig 5). This is because of the following reasons:

- First, the average number of states at each layer for a worker is subject to the number of workers used in the experiments. When the average number of states at each layer for a worker is high, we may increase the number of workers to improve the running performance of the tool. However, up to a certain point, the more workers used, the less busy workers are and the more burden the master needs to handle and communicate with workers, making the running performance not improve. For example, when the number of workers is increased from 16 to 24, running performances do not improve and even becomes a bit worse for the first and third case studies. In addition
- Second, we use `simBatch` and `simBatchH` to calculate the number of needed workers to conduct step

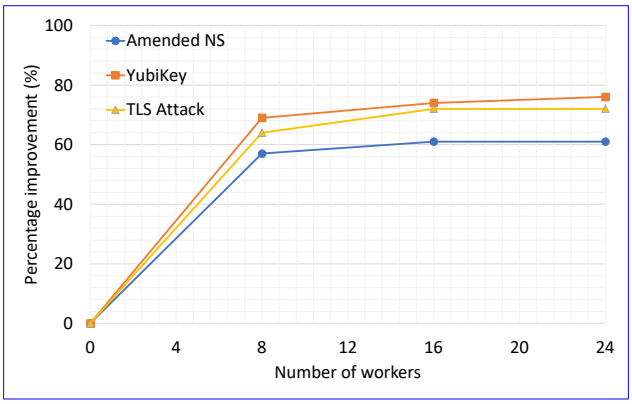


Fig. 5: The percentage of improvement of Par-Maude-NPA-2

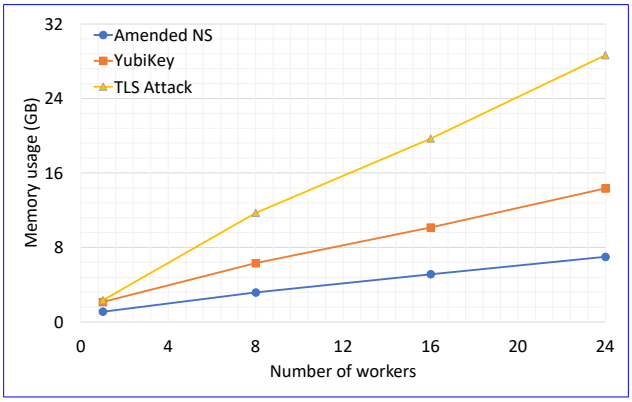


Fig. 6: The memory usage of Par-Maude-NPA-2

(2.1) and step (2.2) in parallel at each layer, respectively. At the very first layers, the number of successor states and history states is still small. Therefore, we may not need to conduct step (2.1) and step (2.2) in parallel. For deeper layers, when the number of successor states and history states becomes larger, step (2.1) and step (2.2) may be conducted in parallel. When we use `simBatch` and `simBatchH`, there is a case in which the number of needed workers is less than the number of workers available. Hence, even if we increase the number of workers, it does not contribute to the running performances. ~~Furthermore~~

- Last, when we parallelize step (2.1), we need to generate (simplified) jobs and then combine every two jobs into one job until there is only one job left. Whenever we combine two (simplified) jobs produced by two workers, we only need one worker to conduct the combination while the other worker is free. Therefore, all workers do not always work at any time. Up to a certain point, even if the number of workers is increased, running performances do not improve.

Regarding memory usage, we measure the maximum resident set size (RSS), the largest memory Par-Maude-NPA-2 has ever used while running experiments for each protocol. When we increase the number of workers, memory usage increases almost linearly (see Fig. 6), although the percentage of improvement increases slowly with 16 and 24 workers (see Fig. 5). The linear increase in memory usage is understandable because the more workers are used at the same time, the more memory is consumed. However, the maximum RSS does not reflect the percentage of improvement of running performance obtained from Par-Maude-NPA-2. With the same maximum RSS, the busier the workers are, the greater the percentage of improvement tends to be.

5.5 The use of `simBatch` and `simBatchH`

As described above~~we use *simBatch* and *simBatchH*~~, we use `simBatch` and `simBatchH`to avoid assigning a light job to a worker. The use of ~~*simBatch* and *simBatchH*~~ `simBatch` and `simBatchH` is applicable when the number of states and history states at a layer is reasonably small. When the number of states and history states at a layer is large, the states at a layer should be divided evenly into each worker to take advantage of parallelization as much as possible. From our experiences when conducting experiments for those protocols with the tool, we set ~~*simBatch* and *simBatchH*~~ `simBatch` and `simBatchH` to 20 and 50 as default, respectively. For example, we need to conduct the transition subsumption in step (2.1) for 20 states. In the worst case, no state is implied by other states in 20 states, Maude-NPA needs to spend about 380 (i.~~g.,~~ ~~$2 \times 19 \times 20 \div 2$~~e., 19×20) computations in which each computation is to check the implication between two states. Hence, 20 states may be the minimum number of states that should be assigned to a worker. Likewise, ~~*simBatchH*~~ `simBatchH` is used in the transition subsumption with history states in step (2.2). There is a trade-off between increasing and decreasing those numbers because it affects the number of jobs as well as the number of workers used at each layer. However, we should use reasonably small values as our default values because we are concerning the minimum number of states that should be assigned to a worker. ~~Nevertheless, we need~~ The values of `simBatch` and `simBatchH` are considered a threat to internal validity in our work because they could affect the evaluation results. To mitigate this threat, we plan to conduct more experiments with different values for ~~*simBatch* and *simBatchH*~~ `simBatch` and `simBatchH` from which we may select ~~better~~ optimal values for them with our tool~~as~~. This is one piece of our future work.

~~In summary,~~

5.6 The necessity of Par-Maude-NPA-2

Par-Maude-NPA aims to improve the running performance of Maude-NPA, enabling the efficient analysis of larger case studies within a reasonable amount of time. Our research group has formally specified and verified Hybrid Post-Quantum TLS in Maude-NPA and Par-Maude-NPA2 [40]. This case study may be the largest one tackled by Maude-NPA so far. We focus on verifying the secrecy property of the ECDH shared secret key established between two honest principals under some assumptions. Par-Maude-NPA-2 successfully found a counterexample of this property within 1,722 hours (about 72 days), while Maude-NPA failed to find it even after a very long time

(about 178 days). Without the use of Par-Maude-NPA-2, Maude-NPA could not have detected the attack due to an excessively long analysis time. This highlights the necessity of Par-Maude-NPA-2 in the formal verification of security protocols.

5.7 Summary

Par-Maude-NPA-2 can improve the running performance of Maude-NPA effectively when dealing with complex case studies in which the number of states located at each layer is considerably large. ~~For the backward narrowing in step (1), the more states located at each layer and the deeper the search space is, the more improvement may be obtained by parallelization. For the transition subsumption in step (2.1) and step (2.2), the more successor states and history states at each layer, the more improvement may also be obtained by parallelization. For~~ For simple case studies, whose verification time is very ~~small~~short, for example, less than 40 seconds in our case studies, we do not need to use Par-Maude-NPA-2, although we still can use it to obtain a result in a reasonably short amount of time. We can see that the verification time for simple case studies is very small and so the use of Par-Maude-NPA-2 is not much different compared to Maude-NPA in terms of verification time. Hence, it is sufficient to use solely Par-Maude-NPA-2 to analyze any cryptographic protocols even when they are simple.

6 Related Work

In addition to Maude-NPA, there are several cryptographic analysis tools for security protocols, such as Athena [10], ProVerif [11], Avispa [12], CL-Atse [13], Scyther [14], Tamarin [15], AKISS [16], DEEPSEC [17], Verifpal [18], and CPSA [19]. Among them, some symbolic tools are described and compared with Maude-NPA as follows.

Scyther [14] is an automatic analyzer that supports both an unbounded number of sessions and a bounded number of sessions, and always terminates. However, it only considers a fixed set of cryptographic primitives consisting of symmetric and asymmetric encryption. Security properties verified in Scyther are mostly trace properties that hold for any execution trace, where secrecy and authentication properties can be expressed. Maude-NPA supports rich cryptographic primitives that are user-definable and it can verify not only trace properties as its natural reachability analysis but also equivalence properties [41], which state that two processes in a security protocol are equivalent when an adversary cannot distinguish the difference between interactions with two processes. Verifying equivalence properties is harder than verifying trace properties because we need to consider the relation between traces instead of a single trace.

Tamarin [15] is a prover that generalizes the backward search used by the Scyther tool and supports an unbounded session model, reasoning modulo equational theories, modeling complex control flow (e.g., loops), and mutable global state. It provides both automatic and interactive modes to construct proofs. However, it often needs some lemmas provided by users to complete its proofs. In Tamarin, a protocol specification is specified by means of multiset rewriting rules, while a property specification is written as a guarded fragment of first-order logic. Each protocol trace corresponds to a multiset rewriting derivation that is the sequences of the labels of the applied rules. Tamarin performs an exhaustive backward search to look for a trace that does not satisfy the property and returns a counterexample as an attack. If no rule can be applied anymore and no counterexample is found, then the protocol satisfies the property. Tamarin can verify trace properties and observational equivalence properties. ~~To make the problem of security protocol verification decidable~~In order to expand the class of protocols and crypto properties accepted by the tool, Tamarin also uses the finite variant property [42] to reduce reasoning modulo an equational theory with respect to a rewrite theory as in Maude-NPA. ~~Basically, Tamarin can support at the same level as~~ For complex protocols, the interactive mode is often used with necessary lemmas when the automatic mode fails to terminate. Maude-NPA ~~in cryptographic protocol analysis . However,~~ is a fully automatic verification tool and it also supports never patterns specified in attack states to cut down the search space, making the formal analysis faster. Moreover, Tamarin also supports multi-threading to speed up its proof search as described in their manual, but how it was done is not mentioned at all. Tamarin, with an automatic mode, actually performs a depth-first search (DFS). It seems that they have tried to parallelize the DFS, while we parallelize the BFS in Maude-NPA~~does not require lemmas from users and it is fully automatic~~ .

ProVerif [11] is an abstraction-based approach to symbolically analyzing cryptographic protocols. The protocol specifications that are specified in an extension of the pi calculus are translated into Horn clauses and the security properties being proved are translated into derivability queries on the Horn clauses. ProVerif uses a resolution algorithm to check whether a fact is derivable from the clauses. If there is no derivation, the property is proved. Otherwise, the derivation found is reconstructed at the pi calculus level as an attack. However, the attack may be spurious because some abstractions are used in Horn clauses. In the case of a false attack, ProVerif cannot conclude anything. ~~However~~Recently, ProVerif has ~~recently~~ been extended with lemmas, axioms, proofs by induction, natural numbers, and temporal queries that can ~~prevent~~ help in dealing with the false attack situation [43]. However, they cannot entirely prevent it and the underlying problem is still undecidable. ProVerif can verify secrecy, authentication, and some observational equivalence properties. Besides, cryptographic primitives can be defined by equations or rewrite rules~~that also need to~~ . To get some decidability results, ProVerif considers not only the finite variant property but also linear equational theories. Meanwhile, Maude-NPA only accepts linear equational theories if they satisfy the finite variant property~~to make the analysis terminate as in Maude-NPA~~. However, ~~it~~ ProVerif does not support associativity, commutativity, and homomorphic properties as Maude-NPA.

DEEPSEC [17] focuses on deciding trace equivalence properties in security protocols, which are specified in a dialect of the applied pi calculus [44]. However, it only supports a bounded number of sessions and cryptographic

primitives are specified by a set of subterm convergent rewrite rules, where the right-hand side of each rule must be a subterm of the left-hand side or a ground term. To guide the decision of equivalence of two processes in cryptographic protocols, DEEPSEC constructs a so-called partition tree, where each node consists of a set of symbolic processes and constraints. Initially, the root node only consists of the two symbolic processes and empty constraints. Given a node, sibling nodes can be constructed based on some rules in DEEPSEC. The partition tree is then constructed in a top-down style. While constructing the partition tree, if there is some node that does not contain both two processes originated from the two beginning processes, DEEPSEC returns an attack; otherwise, there are no attacks. Because sibling nodes are independent, the construction of the subtree from each sibling node can be processed in parallel as follows. DEEPSEC maintains a queue of jobs with a fixed size. It first starts with a breadth-first search from the root node to generate all successor nodes and put them into the queue until reaching the size. Each idle worker can fetch a job to handle and checks if the queue is full. If so the worker starts constructing the entire subtree from the node included in the job; otherwise, it keeps on producing jobs to put into the queue. The way to parallelize DEEPSEC is different from ours because we never generate the entire subtree from a node. It also seems that DEEPSEC does concern visited nodes while constructing the partition tree. ~~To the best of our knowledge,~~

Besides Maude-NPA~~is the first cryptographic security tool parallelizedfor an unbounded number of sessions~~, there are only a few security protocol verification tools that have been parallelized, such as CL-Atse, Tamarin, AKISS, and DEEPSEC, among which only DEEPSEC has well-documented how the parallelization has been done [17] regarding our investigation. Although there are many parallel model checking algorithms for LTL [45], such as DiVinE 3.0 [46], Garakabu2 [47], [48], a multicore extension of SPIN [49], and Parallel $L+1$-DCA2L2MC [50]~~, where DCA2L2MC stands for a divide & conquer approach to leads-to model checking.~~ .

Regarding the efficiency of security protocol verification tools, there have been only a few studies that compare their performances [51], [52], [53], [54], [55]. That is mainly because it requires understanding each tool, each protocol, and desired properties to be able to write in each specific modeling language. For security protocol verification tools dealing with algebraic properties, there are some studies on the comparison [54], [55], but no clear winner is presented in terms of efficiency when it depends on each protocol and desired properties. As one piece of future work, we aim to select some specific protocols and analyze them with their properties using Par-Maude-NPA and the parallelization of Tamarin to compare their performance efficiency.

7 Discussion

It would be ~~(almost) impossible~~ hard to improve the running performance of Maude-NPA by parallelization beyond the parallel version of Maude-NPA described in the present paper, where steps (1) and (2) are conducted in parallel at each layer. Because the transition subsumption is crucial to make Maude-NPA reasonably fast and it may transform an infinite system into a finite one [33], we should synchronize all jobs at the end of each layer and it would not be possible to parallelize multiple layers or entire reachable sub-state spaces from states at a layer as parallel DEEPSEC and parallel $L+1$-DCA2L2MC have ~~been~~ done. It is true that it is worth improving the running performance of any tool for formal analysis and verification of security protocols, such as Maude-NPA. ~~It is also true that (sequential) Maude-NPA has been fully optimized, which means that it would be almost impossible to improve its running performance without relying on parallelization. Parallelization could be one possible optimization technique to improve the running performanceof a tool, however, it is~~ Parallelization is one of the main streams to improve running performance. However, it is also true that it is really tough to improve the running performance of any tool that has already adopted many optimization techniques in a serial way, such as Maude-NPA. Moreover, it ~~also~~ requires a deep and careful analysis of how a tool works in detail in order to parallelize ~~the tool~~it. Our goal is to improve the running performance of Maude-NPA as much as possible by parallelizing some of its tasks. Because the whole task conducted in Maude-NPA cannot be naturally divided into multiple independent tasks and there are some synchronized points at each layer, it would be impossible to gain a few times faster running performance by parallelizing Maude-NPA. Even if we do not gain multiple times faster by parallelization, it must be worth parallelizing a tool if we get some running performance improvement. As ~~we have~~ demonstrated, our tool can obtain better running performances when dealing with complex security protocols that have large state spaces. ~~Therefore, parallelization is used to improve the running performance of~~ Especially there is a case study in which Par-Maude-NPA successfully detected an attack, while Maude-NPA ~~as some optimization techniques in Maude-NPA~~failed to find the attack due to an excessively long analysis time [40].

8 Conclusion

The paper has described a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption are conducted in parallel. A tool has been developed in Maude to support the parallel version with a master-worker model by using meta-interpreters instead of Maude sockets as in our previous work [50]. The paper has also reported on some experiments of various kinds of protocols in which the tool can increase running performances of both Maude-NPA (about 44%) and our previous tool (about 23%) on average for the complex case studies used for experiments where one master and eight workers are used. Especially, the tool can largely improve the running performance of Maude-NPA by 57%, 69%, and 64% for the three case studies, respectively, whose verification time is the largest among the protocols used for experiments, demonstrating its potential to deal with case studies whose state space is large. As one piece of our future work, we should conduct more case studies and use various numbers of workers with the tool to demonstrate its usefulness. ~~Last but not least, the basic techniques used to parallelize Maude-NPA are not~~

~~necessarily specific to Maude-NPA, but could be used to parallelize other tools dedicated to cryptographic protocol analysis, especially TAMARIN, because TAMARIN is the closest to Maude-NPA.~~

ACKNOWLEDGMENTS

This work was supported by JST SICORP Grant Number JPMJSC20C2, Japan, by grant S2018/TCS-4339 (BLOQUES-CM) funded by Comunidad de Madrid co-funded by EIE Funds of the European Union, by grant PID2019-108528RB-C22 (ProCode-UCM) funded by MICIN. S. Escobar has been partially supported by the grant ~~RTI2018-094403-B-C32~~ PID2021-122830OB-C42 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe ~~, by the grant PROMETEO/2019/098 funded by Generalitat Valenciana,~~ and by the grant PCI2020-120708-2 funded by MICIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.

REFERENCES

[1] C. M. Do, A. Riesco, S. Escobar, and K. Ogata, "Parallel maude-npa for cryptographic protocol analysis," in *Rewriting Logic and Its Applications - 14th International Workshop, WRLA 2022*, ser. Lecture Notes in Computer Science, vol. 13252. Springer, 2022, pp. 253–273.

[2] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," *RFC*, vol. 5246, pp. 1–104, 2008.

[3] E. Rescorla, "The transport layer security (tls) protocol version 1.3," *RFC*, vol. 8446, pp. 1–160, 2018.

[4] L. Dong and K. Chen, *Introduction of Cryptographic Protocols*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–12.

[5] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," *Inf. Process. Lett.*, vol. 56, no. 3, p. 131–133, Nov. 1995.

[6] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, p. 993–999, Dec. 1978.

[7] A. Satapathy and J. Livingston, "A comprehensive survey on ssl/ tls and their vulnerabilities," *International Journal of Computer Applications*, vol. 153, pp. 31–38, 11 2016.

[8] Amazon Web Services, "Hybrid key exchange in TLS 1.3," https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design, 2023, [Online; accessed 24-May-2023].

[9] S. Escobar, C. A. Meadows, and J. Meseguer, "Maude-npa: Cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, ser. Lecture Notes in Computer Science, A. Aldini, G. Barthe, and R. Gorrieri, Eds., vol. 5705. Springer, 2007, pp. 1–50.

[10] D. X. Song, "Athena: a new efficient automatic checker for security protocol analysis," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999, pp. 192–202.

[11] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, 2001, pp. 82–96.

[12] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA tool for the automated validation of internet security protocols and applications," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 281–285.

[13] M. Turuani, "The cl-atse protocol analyser," in *Term Rewriting and Applications*, F. Pfenning, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 277–286.

[14] C. J. Cremers, "The scyther tool: Verification, falsification, and analysis of security protocols," in *Proceedings of the 20th International Conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 414–418.

[15] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701.

[16] R. Chadha, V. Cheval, c. Ciobâcă, and S. Kremer, "Automated verification of equivalence properties of cryptographic protocols," *ACM Trans. Comput. Logic*, vol. 17, no. 4, sep 2016. [Online]. Available: https://doi.org/10.1145/2926715

[17] V. Cheval, S. Kremer, and I. Rakotonirina, "DEEPSEC: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 529–546.

[18] N. Kobeissi, G. Nicolas, and M. Tiwari, "Verifpal: Cryptographic protocol analysis for the real world," in *Progress in Cryptology – INDOCRYPT 2020*, K. Bhargavan, E. Oswald, and M. Prabhakaran, Eds. Cham: Springer International Publishing, 2020, pp. 151–202.

[19] J. D. Ramsdell, "Cryptographic protocol analysis and compilation using cpsa and roletran," in *Protocols, Strands, and Logic*, D. Dougherty, J. Meseguer, S. A. Mödersheim, and P. Rowe, Eds. Cham: Springer International Publishing, 2021, pp. 355–369.

[20] D. Baelde, S. Delaune, and L. Hirschi, "Partial order reduction for security protocols," in *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, ser. LIPIcs, L. Aceto and D. de Frutos-Escrig, Eds., vol. 42. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 497–510.

[21] G. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[22] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[23] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[24] F. Fabrega, J. Herzog, and J. Guttman, "Strand spaces: why is a security protocol correct?" in *Proceedings. 1998 IEEE Symposium on Security and Privacy*, 1998, pp. 160–171.

[25] S. Escobar, C. Meadows, and J. Meseguer, "A rewriting-based inference system for the nrl protocol analyzer and its meta-logical properties," *Theor. Comput. Sci.*, vol. 367, no. 1, p. 162–202, Nov. 2006.

[26] ——, "State space reduction in the maude-nrl protocol analyzer," in *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ser. ESORICS '08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 548–562.

[27] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4350.

[28] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. C. Winkler, "An introduction to OBJ3," in *1st CTRS Workshop*, ser. Lecture Notes in Computer Science, vol. 308. Springer, 1987, pp. 258–263.

[29] J. Meseguer, "Twenty years of rewriting logic," *J. Log. Algebraic Methods Program.*, vol. 81, no. 7-8, pp. 721–781, 2012.

[30] K.-L. Wright and K. Gopalan, "Performance analysis of interprocess communication mechanisms," Binghamton University, Tech. Rep. TR-20070820, Tech. Rep., 2007.

[31] S. Escobar, R. Sasse, and J. Meseguer, "Folding variant narrowing and optimal variant termination," in *Rewriting Logic and Its Applications*, P. C. Ölveczky, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 52–68.

[32] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, "Reflection, metalevel computation, and strategies," in *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds. Springer, 2007, vol. 4350, pp. 419–458.

[33] S. Escobar and J. Meseguer, "Symbolic model checking of infinite-state systems using narrowing," in *Term Rewriting and Applications*, F. Baader, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 153–168.

[34] F. Yang, S. Escobar, C. A. Meadows, J. Meseguer, and S. Santiago, "Strand spaces with choice via a process algebra semantics," in *Proceedings of the 18th International Symposium on Principles and*

Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016, J. Cheney and G. Vidal, Eds. ACM, 2016, pp. 76–89.

[35] S. Escobar, C. A. Meadows, J. Meseguer, and S. Santiago, "Symbolic protocol analysis with disequality constraints modulo equational theories," in *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, ser. Lecture Notes in Computer Science, C. Bodei, G. L. Ferrari, and C. Priami, Eds., vol. 9465. Springer, 2015, pp. 238–261.

[36] A. González-Burgueño, S. Santiago, S. Escobar, C. A. Meadows, and J. Meseguer, "Analysis of the pkcs#11 API using the maude-npa tool," in *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, ser. Lecture Notes in Computer Science, L. Chen and S. Matsuo, Eds., vol. 9497. Springer, 2015, pp. 86–106.

[37] ——, "Analysis of the IBM CCA security API protocols in maude-npa," in *Security Standardisation Research - First International Conference, SSR 2014, London, UK, December 16-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. Chen and C. J. Mitchell, Eds., vol. 8893. Springer, 2014, pp. 111–130.

[38] S. Escobar, D. Kapur, C. Lynch, C. A. Meadows, J. Meseguer, P. Narendran, and R. Sasse, "Protocol analysis in maude-npa using unification modulo homomorphic encryption," in *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, P. Schneider-Kamp and M. Hanus, Eds. ACM, 2011, pp. 65–76.

[39] A. González-Burgueño, D. Aparicio, S. Escobar, C. A. Meadows, and J. Meseguer, "Formal verification of the yubikey and yubihsm apis in maude-npa," *CoRR*, vol. abs/1806.07209, 2018.

[40] D. D. Tran, C. M. Do, S. Escobar, and K. Ogata, "Hybrid post-quantum tls formal analysis in maude-npa and par-maude-npa," in *Subbmited for publication*, S. Akleylek, S. Escobar, K. Ogata, and A. Otmani, Eds., 2023.

[41] S. Santiago, S. Escobar, C. A. Meadows, and J. Meseguer, "A formal definition of protocol indistinguishability and its verification using maude-npa," in *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, ser. Lecture Notes in Computer Science, S. Mauw and C. D. Jensen, Eds., vol. 8743. Springer, 2014, pp. 162–177.

[42] H. Comon-Lundh and S. Delaune, "The finite variant property: How to get rid of some algebraic properties," in *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. Giesl, Ed., vol. 3467. Springer, 2005, pp. 294–307.

[43] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 69–86.

[44] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, C. Hankin and D. Schmidt, Eds. ACM, 2001, pp. 104–115.

[45] J. Barnat, V. Bloemen, A. Duret-Lutz, A. Laarman, L. Petrucci, J. van de Pol, and E. Renault, "Parallel model checking algorithms for linear-time temporal logic," in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 457–507.

[46] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser, "DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs," in *CAV 2013*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 863–868.

[47] W. Kong, L. Liu, T. Ando, H. Yatsu, K. Hisazumi, and A. Fukuda, "Facilitating multicore bounded model checking with stateless explicit-state exploration," *Comput. J.*, vol. 58, no. 11, pp. 2824–2840, 2015.

[48] W. Kong, G. Hou, X. Hu, T. Ando, K. Hisazumi, and A. Fukuda, "Garakabu2: an SMT-based bounded model checker for HSTM designs in ZIPC," *J. Inf. Sec. Appl.*, vol. 31, pp. 61–74, 2016.

[49] G. J. Holzmann and D. Bosnacki, "The design of a multicore extension of the SPIN model checker," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 659–674, 2007.

[50] C. M. Do, Y. Phyo, A. Riesco, and K. Ogata, "A parallel stratified model checking technique/tool for leads-to properties," in *2021 7th International Symposium on System and Software Reliability (ISSSR)*, 2021, pp. 155–166.

[51] L. Viganò, "Automated security protocol analysis with the avispa tool," *Electronic Notes in Theoretical Computer Science*, vol. 155, pp. 61–86, 2006, proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066106001897

[52] M. Cheminod, I. Bertolotti, L. Durante, R. Sisto, and A. Valenzano, "Experimental comparison of automatic tools for the formal analysis of cryptographic protocols," 07 2007, pp. 153 – 160.

[53] C. J. F. Cremers, P. Lafourcade, and P. Nadeau, *Comparing State Spaces in Automatic Security Protocol Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 70–94. [Online]. Available: https://doi.org/10.1007/978-3-642-02002-5_5

[54] P. Lafourcade, V. Terrade, and S. Vigier, "Comparison of cryptographic verification tools dealing with algebraic properties," in *Formal Aspects in Security and Trust*, P. Degano and J. D. Guttman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 173–185.

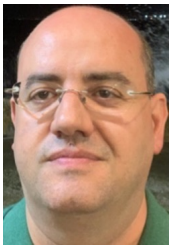
[55] P. Lafourcade and M. Puys, "Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties," in *Foundations and Practice of Security*, J. Garcia-Alfaro, E. Kranakis, and G. Bonfante, Eds. Cham: Springer International Publishing, 2016, pp. 137–155.

Canh Minh Do received his M.S. and Ph.D. degrees in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2019 and 2022, respectively.

He is currently ~~a researcher~~ an assistant professor at JAIST. He has been working on the parallelization of formal verification tools used in formal methods.

Adrián Riesco received his M.S. and Ph.D. degrees in information technology from the Universidad Complutense de Madrid.

He is currently an associate professor at the Universidad Complutense de Madrid. His research interests are declarative debugging, test case generation, distributed applications, rewriting logic, unification, and narrowing.



Santiago Escobar received his M.S. and Ph.D. degrees in information technology from Universitat Politècnica de València.

He is currently an associate professor at Universitat Politècnica de València. His research interests are declarative debugging, term rewriting, rewriting and narrowing strategies, security, and model checking.

Kazuhiro Ogata received his B.S, M.S. and PhD degrees in engineering from Keio University in 1990, 1992 and 1995, respectively.

He is currently a professor at the Japan Advanced Institute of Science and Technology (JAIST). His research focuses on applications of formal methods to systems, such as distributed systems and security protocols.

TABLE 2: Experimental results of Maude-NPA and Par-Maude-NPA-2.

Protocol	Attack ID	Result at depth	Maude-NPA (seconds)	Par-Maude-NPA-2 (seconds)	P(%)	States/ Layer/ Worker
1. Symmetric Key Protocols						
Amended Needham Schroeder	0	✗ / 7	4589	**1968**	57	11.11
Carlsen Secret Key Initiator	0	✗ / 5	224	**116**	48	3.73
Denning Sacco	0	✓ / 11	35	**30**	14	0.43
Diffie Hellman Key Exchange	0	✗ / 11	284	**114**	60	1.56
	1	✗ / 12	287	**104**	64	1.45
	2	✓ / 13	35	**23**	35	0.32
ISO-5 Pass Authentication	0	✗ / 5	102	**55**	46	2.1
Kao-Chow RA	0	✗ / 4	52	**29**	45	2
Kao-Chow RAHK	0	✗ / 4	**4**	14	-72	0.19
Kao-Chow RAT	0	✗ / 4	114	**67**	41	1.94
Otway-Rees	0	✗ / 4	73	**43**	41	2.16
Secret 06	0	✗ / 2	**1.7**	6.9	-75	0.38
Secret 07	0	✗ / 4	**2.6**	9.2	-72	0.28
Wide Mouthed Frog	0	✗ / 3	16	**15**	7	1.92
Woo and Lam Authentication	0	✗ / 4	**1.4**	8.3	-83	0.28
Yahalom	0	✗ / 4	45	**28**	39	1.91
2. Homomorphism Protocols						
Needham Schroeder Lowe ECB	0	✗ / 7	74	**45**	39	1.11
3. Exclusive OR Protocols						
Needham Schroeder Lowe XOR	0	✗ / 8	**10.2**	13.9	-26	0.31
SK3	0	✓ / 3	**4.2**	12	-65	0.17
TMN ltv-F-tmn-asy	0	✗ / 5	157	**38**	76	0.88
WIRED ltv-C-wep-asy	0	✓ / 5	**14.4**	20.6	-30	0.15
WIRED ltv-C-wep-variant	0	✓ / 5	**15.6**	23	-32	0.15
4. API Protocols						
YubiKey	0	✗ / 9	**3.5**	12.7	-72	0.17
	1	✓ / 7	93825	**28989**	69	5.13
	21	✓ / 8	342	**135**	61	0.8
	3	✓ / 7	13093	**4226**	68	3
YubiHSM attack(d)	0	✗ / 9	843	**405**	52	2.38
5. PKCS Protocols						
PKCS11 a1-noComp	0	✗ / 4	24.8	**23.6**	5	0.81
PKCS11 a2-noComp	0	✗ / 6	70	**45**	36	0.75
PKCS11 a3-noComp	0	✗ / 6	296	**165**	44	1.6
PKCS11 a4-noComp	0	✗ / 7	63	**39**	38	0.88
PKCS11 a5-noComp	0	✗ / 9	382	**225**	41	1.82

TABLE 3: Experimental results of Maude-NPA and Par-Maude-NPA-2.

Protocol	Attack ID	Result at depth	Maude-NPA (seconds)	Par-Maude-NPA-2 (seconds)	P2(%)	States/ Layer/ Worker
6. Choice Protocols						
encryption mode	0	✗ / 4	**3.2**	10.7	-70	0.28
	1	✗ / 4	**8.6**	11.8	-27	0.78
	2	✓ / 10	68	**40**	41	1.1
	3	✓ / 11	137	**56**	59	1.61
rock paper scissors	0	✓ / 9	126	**53**	58	1.81
	1	✓ / 1	**0.4**	5.3	-93	0.13
	2	✓ / 2	**1**	6.6	-85	0.38
TLS regular	0	✗ / 3	**6.7**	13.8	-51	0.17
TLS attack	0	? / 11	8695	**3151**	64	3.15
7. Distance-Bounding Protocols						
brands chaum	1	✓ / 4	**6.2**	11.8	-47	0.25
	2	✗ / 6	**16.2**	17.1	-5	0.29
CRCS	1	✓ / 9	767	**292**	62	0.75
	2	✗ / 8	122	**83**	32	0.42
H&K	1	✓ / 5	16.8	**15.4**	8	0.35
	2	✓ / 2	**1.2**	7.1	-84	0.13
MAD	1	✓ / 9	175	**97**	45	0.67
	2	✗ / 6	967	**396**	59	2.42
Meadows v1-DH	1	✓ / 4	**1.6**	9.1	-82	0.13
	2	✓ / 8	**32.2**	32.8	-2	0.28
Meadows v2-DH	1	✓ / 4	**1.7**	9.1	-81	0.13
	2	✗ / 3	**2.5**	8.9	-72	0.17
Munilla	1	✓ / 7	186	**67**	64	1.45
	2	✓ / 4	**6.3**	12.9	-51	0.19
Swiss Knife	1	✓ / 4	**6.7**	12.2	-45	0.25
	2	✓ / 4	26.9	**25.5**	5	0.38
TREAD	1	✓ / 4	**6.4**	12.2	-47	0.25
	2	✗ / 4	**5.2**	11.8	-56	0.25

TABLE 4: The effectiveness of the use of meta-interpreters and the parallelization of step (2.1) and step (2.2).

Protocol	Attack ID	Result at depth	Maude-NPA (seconds)	Par-Maude-NPA-1 (seconds)	P1(%)	Par-Maude-NPA-1* (seconds)	P1*(%)	Par-Maude-NPA-2 (seconds)	P2(%)
Amended NS	0	✗ / 7	4589	**2822**	39	**2616**	43	**1968**	57
YubiKey	1	✓ / 7	93825	**65295**	30	**59174**	37	**28989**	69
TLS attack	0	? / 11	8695	**6997**	20	**4366**	50	**3151**	64

TABLE 5: Par-Maude-NPA-1 and Par-Maude-NPA-2 with various numbers of workers.

Protocol	Attack number	Result at depth	Maude-NPA (seconds)	#Workers	Par-Maude-NPA-1 (seconds)	P1(%)	Par-Maude-NPA-2 (seconds)	P2(%)	States/ Layer/ Worker
Amended NS	0	✗ / 7	4589	8	**2822**	39	**1968**	57	11.11
				16	**2652**	42	**1787**	61	5.55
				24	**2646**	42	**1791**	61	3.7
YubiKey	1	✓ / 7	93825	8	**65295**	30	**28989**	69	5.13
				16	**61365**	35	**24756**	74	2.56
				24	**60937**	35	**22088**	76	1.71
TLS attack	0	? / 11	8695	8	**6997**	20	**3151**	64	3.15
				16	**6961**	20	**2393.5**	72	1.57
				24	**7194**	17	**2392.8**	72	1.05