PFMC: A Parallel Symbolic Model Checker for Security Protocol Verification

Alex James, Alwen Tiu, and Nisansala Yatapanage

School of Computing, The Australian National University

Abstract. We present an investigation into the design and implementation of a parallel model checker for security protocol verification that is based on a symbolic model of the adversary, where instantiations of concrete terms and messages are avoided until needed to resolve a particular assertion. We propose to build on this naturally lazy approach to parallelise this symbolic state exploration and evaluation. We utilise the concept of *strategies* in Haskell, which abstracts away from the low-level details of thread management and modularly adds parallel evaluation strategies (encapsulated as a monad in Haskell). We build on an existing symbolic model checker, OFMC, which is already implemented in Haskell. We show that there is a very significant speed up of around 3-5 times improvement when moving from the original single-threaded implementation of OFMC to our multi-threaded version, for both the Dolev-Yao attacker model and more general algebraic attacker models. We identify several issues in parallelising the model checker: among others, controlling growth of memory consumption, balancing lazy vs strict evaluation, and achieving an optimal granularity of parallelism.

1 Introduction

A security protocol describes a sequence of actions and message exchanges between communicating partners in a networked system, in order to achieve certain security goals, such as authentication and secrecy. The analysis of security protocols against a stated security property is a challenging problem, not just from a computational complexity perspective (e.g., the problem of establishing the secrecy property of a protocol is undecidable in general [16]), but also from a protocol designer perspective, since proofs of security properties are dependent on the adversary model, which can be challenging to precisely formalise.

A commonly used adversary model is the Dolev-Yao model [15]. The Dolev-Yao model assumes that the attacker controls the network, and hence will be able to intercept, modify and remove messages. However, the attacker is also assumed to be unable to break the basic cryptographic primitives used in the protocol. For example, an encrypted message can only be decrypted by the attacker if and only if the attacker possesses the decryption key. The Dolev-Yao model may be further extended by adding various abstract algebraic properties (e.g., theories for modelling exclusive-or, or Abelian groups in general). In the context

of protocol analysis, the Dolev-Yao model and/or its algebraic extensions are sometimes referred to as the symbolic model.

In this paper, we are concerned with verifying protocols with a bounded number of sessions in the symbolic model, and we restrict to only proving reachability properties, i.e., those properties that can be expressed as a predicate on a single trace of a protocol run, such as secrecy and authentication. Bounded verification aims primarily at finding attacks, but even for a small number of sessions, the complexity of finding attacks is still very high, e.g., NP-complete for the standard Dolev-Yao model [22]. Another interesting use of bounded verification is to lift the result of such verification to the unbounded case, for a certain class of protocols [3]. A related work by Kanovich et al. [17] on a bounded-memory adversary (which implies bounded number of sessions) also points to the fact that many attacks on existing protocols can be explained under a boundedmemory adversary. These suggest that improving the performance of bounded protocol verifiers may be a worthwhile research direction despite the prevalence of protocol verifiers for unbounded sessions such as Proverif [9] and Tamarin [20].

A bounded-session protocol verifier works by state exploration (search), so naturally we would seek to improve its performance by parallelising the search. Given the ubiquity of multicore architecture in modern computing, it is quite surprising that very few protocol verifiers have attempted to benefit from the increase in computing cores to speed up their verification efforts. An important consideration in our attempt is avoiding excessive modification of the (implementation of) decision procedures underlying these verifiers, as they are typically complex and have been carefully designed and optimised. This motivates us to consider a language where parallelisation (of a deterministic algorithm) can be added as an effect – Haskell parallelisation monads (e.g., [19]) fit this requirement very well. In this work, we use OFMC, which is implemented in Haskell, as the basis of our study, looking at ways to introduce parallelisation without changing the underlying implementation of its decision procedures. This will hopefully provide us with a recipe for applying similar parallelisation techniques to other protocol verifiers written in Haskell, notably Tamarin.

We have implemented a parallel version of OFMC [21], which we call PFMC, in Haskell. We show that PFMC significantly improves the performance of OFMC, with a speed-up of around 3-5 times faster than OFMC, when run on 4-16 cores. This promising result allows us to push the boundary of bounded protocol verification. As part of the implementation of PFMC, we have designed what we believe are several novel parallel evaluation strategies for buffered (search) trees in the context of symbolic model checking for protocol verification, which allows us to achieve parallelism with a generally constant memory residency.

Related work. Currently, there are not many major protocol verifiers that explicitly support parallelisation. The only ones that we are aware of are the DEEPSEC prover [12] and the Tamarin prover [20]. DEEPSEC uses a processlevel parallelisation, i.e., each subtask in the search procedure is delegated to a child process and there is no support for a more fine-grained thread-level parallelisation. Tamarin is implemented in Haskell, which has a high-level modular way of turning a deterministic program into one which can be run in parallel, sometimes called semi-explicit parallelisation. This is the method that we will adopt, as it seems like the most straightforward way to gain peformance with minimal effort. As far as we know, there has been no published work on systematically evaluating the performance of Tamarin parallelisation; some preliminary investigations into its parallelisation performance is given in Appendix C. Some of the established protocol verifiers such as Proverif [9], DEEPSEC [12], SATE-QUIV [13], SPEC [24] or APTE [11] are implemented in OCaml, and the support for multicore is not as mature as other languages. For an overview of protocol verification tools, see a recent survey article by Barbosa et. al. [6].

Outline. The rest of the paper is organised as follows. Section 2 provides an overview of the state transition system arising from symbolic protocol analysis. Section 3 gives a brief overview of Haskell parallelisation features. Section 4 presents the parallelisation strategies implemented in PFMC and the evaluation of their performance on selected protocol verification problems. Section 5 concludes and discusses future directions. The full source code of PFMC is online.¹

2 Protocol specifications and transition systems

There are a variety of approaches for modelling protocols and their transition systems, such as using multiset rewriting [10], process calculus [1], strand spaces [23] or first-order Horn clauses [8]. OFMC supports the modelling language IF (Intermediate Format), but it also supports a more user-friendly format known as AnB (for *Alice and Bob*). The AnB syntax can be translated to the IF format, for which a formal semantics is given in [2]; we refer the reader to that paper.

The operational semantics of OFMC is defined in terms of strand spaces. One can think of a strand as a sequence of steps that an agent takes. The steps could be a sending action, a receiving action, checking for equality between messages and special events (that may be used to prove certain attack predicates). We use the notation $A_1 \parallel \ldots \parallel A_n$ to denote *n* parallel strands, A_1, \ldots, A_n , that may be associated with some agents. A *state* is a triple consisting of a set of strands of honest agents, a strand for the attacker, and a set of events which have occurred. The attacker strand consists of the messages the attacker receives by intercepting communication between honest agents, and the messages the attacker synthesises and sends to honest agents. OFMC represents these strands symbolically. For example, the messages the attacker synthesises are initially represented as variables and *concretised* as needed when agents check for equality between messages. The attacker strand is represented as a set of deducibility contraints, and the transition relation must preserve satisfiability of these contraints.

In OFMC, the state transition relation is defined from an adversary-centric view. This means in particular that what matters in the transition is the update to the attacker's knowledge, and the only way to update the attacker's knowledge is through the output actions of the honest agents. Therefore, it makes sense to

¹ https://gitlab.anu.edu.au/U4301469/pfmc-prover

define a state transition as a combination of input action(s) that trigger an output action from an honest agent, rather than using each individual action to drive the state transition. Due to the presence of parallel composition protocol specifications, the (symbolic) state transitions can generate a large number of possible traces, due to the interleaving of parallel strands. The search procedure for OFMC is essentially an exploration of the search tree induced by this symbolic transition system.

3 Haskell parallelisation strategies

This section provides a very brief overview of Haskell parallelisation features and discusses some initial unsuccessful attempts (in the sense that we did not achieve meaningful improvements) to parallelise OFMC, to motivate our designs in Section 4. We use the semi-explicit parallelisation supported by Haskell. We assume the reader is familiar with the basics of Haskell programming, and will only briefly explain the parallelisation approach we use here.

In the semi-explicit parallelism approach in Haskell, the programmer specifies which parts of the code should be paralellised, using primitives such as par. The statement x par y is semantically identically to y. However, the former tells the Haskell runtime system (RTS) to create a worker to evaluate x and assigns it to an available thread to execute. In Haskell terminology, such a unit of work is called a *spark*. The programmer does not need to explicitly create and destroy threads, or schedule the sparks for execution. The RTS uses a *work-stealing* scheduling to execute sparks opportunistically. That is, each available core keeps a queue of sparks to execute and if its queue is empty, it looks for sparks in other cores' queues and 'steals' their sparks to execute. This should ensure all available core so not need to be concerned with low-level details of thread management and scheduling, as they are all handled automatically by the Haskell runtime system.

Determining the granularity of parallelisation. There are three main subproblems in the search for attacks that we examined for potential parallelisation:

- checking the solvability of a constraint system,
- enumerating all possible next states from the current state and
- the overall construction of the search tree itself.

The search for the solutions for a constraint problem can itself be a complex problem, depending on the assumed attacker model. Thus it may seem like a good candidate to evaluate in parallel. However, it turns out that when verifying real-world protocols, most of the constraints generated are simple, and easily solvable sequentially. Attempting to parallelise this leads to worse performance than executing them sequentially, as it ends up creating too many lightweight sparks. It may be the case that as the number of sessions grows, the constraints generated become larger the deeper down the search tree, so at a deeper node in the search tree, such a parallelisation might be helpful. However, to reach that stage, there would likely be a lot of simple constraints that need to be solved for which the overhead of paralellisation outweighs its benefit.

Our next attempt was to parallelise the process for enumerating successor states, which involves solving the intruder constraint problem. This led to some improved performance, but was harder to scale up, as it still created too many sparks, many of which ended up being garbage collected, a sign that there were too many useless sparks. The final conclusion seems to be that focussing on parallelising the construction of the search tree, executing the constraint solving and the successor state enumeration sequentially, produces the most speed up.

Lazy evaluation and parallelism. Haskell by default uses a lazy evaluation strategy in evaluating program expressions. By default, functions in Haskell are evaluated to *weak head normal form* (WHNF). Haskell provides libraries to force a complete evaluation of a program expression, e.g. via the **force** function. This, however, needs to be used with extreme care as it can create unnecessary computation and a potential termination issue.

One advantage of lazy evaluation, when parallelising a search algorithm, is that it allows one to decouple the search algorithm and the parallelisation strategy. In our case, we can implement the search algorithm as if it is proceeding sequentially, constructing a potentially infinite tree of states, without considering termination, as each node will be evaluated lazily only when needed, i.e. when the attack predicate is evaluated against the states in the search tree. Thus, at a very high level, given a function f that constructs a search tree sequentially, and a strategy s for parallelisation, the sequential execution of f can be turned into a parallel execution using the composition:

(withStrategy s) \circ f

Applying this strategy to OFMC, we applied a custom parallelisation strategy (see Section 4.1) to the construction of the search tree. The search algorithm itself does not make any assumption about termination of the search. It does, however, allow the user to specify the depth of search, so any nodes beyond a given depth will not be explored further.

Profiling the runtime behaviour of the parallelisation strategy revealed that a significant amount of time is spent on garbage collection. When searching at a depth d, OFMC keeps the subtrees at depth > d, which end up being garbage collected, as they are not needed in the final evaluation of the attack predicate. A solution is to prune the search tree prior to evaluating the predicate on the nodes. This seems to significantly reduce the memory footprint of the program.

4 Parallel Strategies for Search Trees

We now present a number of parallelisation strategies implemented in PFMC. The actual implementation contains more experimental parallelisation strategies not covered here, as they did not seem to offer much improvements over the main strategies presented here. We note that it is possible to directly benefit from multicore support for the Haskell runtime by compiling the program with the **-threaded** option. However, doing so results in worse performance compared to the single-threaded version, at least in the case for OFMC.

A main difficulty in designing an efficient parallelisation strategy in the case of OFMC is that the branching factor of a given node in the search tree is generally unbounded. Based on our profiling of the search trees of some sample protocols, the branching factor is highly dependent on the number of sessions of the protocol, the assumed intruder model (e.g., constraint solving for some algebraic theories can yield a variable number of solutions), and the depth of the search tree. This makes it difficult to adapt general parallel strategies for bounded trees such as the ones discussed in [25].

The source code for each strategy discussed here can be found in the file src/TreeStrategies.hs in the PFMC distribution.

4.1 parTreeBuffer: a buffered parallel strategy for search trees

A naive parallelisation strategy for OFMC is to simply spark the entire search tree, i.e., for each node in the search tree, we create a spark and evalute it eagerly in parallel. In our tests, it quickly exhausted the available memory in our test server for large benchmark problems, so it does not scale well. Haskell (through the Control.Parallel.Strategies library) provides two ways to avoid creating a large number of sparks simultaneously, via a *chunking* strategy (parChunk), and a *buffered* strategy (parBuffer). They are however restricted to lists. The chunking strategy, as the name suggests, attempts to partition the input list into chunks of a fixed size, and executes all chunks in parallel, but keeping the sequential execution within each chunk. The buffered strategy attempts to stall the creation of sparks, by first creating *n* sparks (for a given value *n*), and then *streaming* the sparks one at a time as needed.

It was not clear what would be an equivalent of parChunk applied to potentially unbounded search trees. In an initial attempt, we tried to flatten the search tree to a list, use the chunking strategy for lists, and 'parse' the list back to a tree. This did not produce the desired effect. We instead designed an approximation of a buffered strategy, but applied to search trees, by controlling the depth of sparking, which we call *par-depth*. For example, for a *par-depth* of 2, all nodes at depth less than or equal to 2 will be sparked, and anything beyond depth 2 will be suspended – by returning their WHNF immediately. The sparks created for nodes that have only been evaluated to WHNF will not attempt to create more sparks for the subtrees. When a suspended node is required by the main thread (e.g., when it needs to be evaluated against a security property), it will trigger another round of sparking (up to *par-depth*)). Figure 1 shows a situation where some of the left-most nodes (marked with the green color) have completed their tasks, and the main thread is starting to query the next suspended node (grey node). This triggers sparking of a subtree of depth 2. As can be seen, the strategy essentially sparks a chunk of seven nodes at a time, in a depth-first manner. The grey nodes represent those that have been only evaluated to WHNF. The vellow nodes represent nodes that may be either finished performing their task, or waiting for results from their child nodes.



Fig. 1. A buffered search tree.

```
data Tree a = Node a [Tree a]
        parTreeBuffer :: Int \rightarrow Int \rightarrow Strategy a \rightarrow Strategy (Tree a)
2
        parTreeBuffer _ _ strat (Node x []) = do
3
          \texttt{y} \leftarrow \texttt{strat} \texttt{x}
          return (Node y [])
        parTreeBuffer 0 n strat (Node x 1) = do
6
          v \leftarrow \texttt{strat} x
          l' ← parBuffer 50 (parTreeBuffer n n strat) l
8
          return (Node y l')
9
        parTreeBuffer !m n strat (Node x 1) = do
          y \leftarrow strat x
11
          l' \leftarrow parList (parTreeBuffer (m-1) n strat) l
          return (Node v l')
```

Fig. 2. A buffered parallel strategy for search trees

This example shows a balanced binary tree. In reality, the search tree may not be balanced, and the branching factor can vary from one node to another. It may be theoretically possible to create a search tree that is infinitely branching, so our strategy cannot completely guarantee that the number of sparks at any given time would be bounded. Indeed, based on our profiling of search trees for a number of benchmarks in OFMC, the search trees can be quite unbalanced, and the branching factors vary greatly between different depths of the search tree. For three sessions of the Kerberos protocol, for example, the branching factors seem to congregate at two extremes: between 1-3 branches at the lower-end versus 40-50 branches at the higher end. In our experiments, we witnessed a relatively constant memory consumption throughout the execution of the benchmark problems up to three sessions. Going beyond four sessions, for large protocols such as Kerberos, the longer the benchmark is executed, the memory consumption may grow substantially, to the point that the entire system memory can be exhausted. Nevertheless, this buffered strategy is simple enough and serves as a good starting point in investigating various trade-offs between the number of cores, the degree of parallelisation and the memory consumption.

Our buffered strategy is implemented as shown in Figure 2. The difference between sparking new nodes and stalling the sparking lies in the use of parBuffer vs. parList: the former evaluates a node into WHNF and returns without creating a new spark, whereas the latter would spark the entire list.



Fig. 3. Profiling workload distributions: (a) overall and (b) an individual activity.

To evaluate the performance of the parBufferTree strategy, we selected three benchmark problems: a flawed version of Google Single Sign-On (SSO) protocol [5], the TLS protocol and a basic version of the Kerberos protocol. All experiments were performed on a server with 96 Intel(R) Xeon(R) Gold 6252 physical cores at 2.10GHz (192 logical cores), and 196GB of RAM. We show some details here for the simplified versions of Google SSO and Kerberos protocols; further details are available in the appendix. For each experiment, we specified the number of sessions and the depth of the search, and the *par-depth* for the clustering of parallel search. The protocols used for these experiments were all specified in the AnB format. These experiments were restricted to a *par-depth* of 3, which seems to strike a reasonable balance between performance and memory consumption. For a protocol with n steps per session, the length of the run of k concurrent sessions of the protocol is bounded by n * k, which corresponds to the depth of search. Therefore, to prove the security of a protocol with k steps for n sessions, the depth of the search must be at least n * k.

To check whether our parallelisation strategy did indeed distribute the work to multiple cores, we perform a profiling of runtime workload distributions among different cores. For this test, we used the Google Single Sign-On (SSO) protocol formalisation which comes with the OFMC distribution. The runtime profiles (Figure 3) were generated using the **threadscope** software [18]. Figure 3a shows the overall profile. The green 'bar' on the top is the overall workload of all cores combined, and the bars below (labelled HEC 0 - HEC 3) correspond to the activities for each core. The green bars represent the actual workload of the program being run. The orange bars denote time spent on garbage collection and the light orange bars represent idle time. As we can see in the figure, the work is distributed evenly across all cores, but there are gaps between activities. Figure 3b shows an individual activity in more detail. The workload component is only slightly longer than the combined GC and idle time. As we will see later, this ratio of workload vs overhead (GC + idle time) is consistently observed throughout our benchmarks.

For the Kerberos protocol, we witnessed a huge increase in the number of states to verify. For 3 sessions and a search depth of 10, using a naive (unbufferred) strategy (e.g. parTree), the verifier occasionally entered a state where it consumed a huge amount of memory (close to 190GB) and the program was terminated by the operating system. The problem became worse when moving to 4 sessions (with a search depth of 12); the memory consumption rose to around

Core#	Total (elapsed) (s)	Total (CPU) (s)	GC (s)	MUT (s)	Mem. res. (MB)
1	7990.432	7989.687	4539.233	3451.164	255.4
2	4992.650	9967.061	3324.002	1668.644	3609.8
4	2891.620	11422.235	1989.502	902.115	3954.3
6	2123.020	12443.051	1483.549	639.460	4122.6
8	1753.690	13493.416	1232.658	521.021	3930.3
10	1627.630	15456.142	1162.774	464.846	3861.1
12	1679.090	18870.687	1219.576	459.509	3829.5
14	1634.090	21193.560	1197.041	437.042	4120.1
16	1563.200	22944.665	1147.829	415.366	4204.5

Table 1. Kerberos protocol verification for 3 sessions and search depth 10.



Fig. 4. Execution time for the Kerberos protocol for 3 sessions with (a) a search depth of 10 and (b) a search depth of 18.

130GB within 3 minutes of runtime. This is thus an example that shows the advantage of the buffered search tree strategy in PFMC.

For this case study, we performed two sets of experiments. The first experiment used 3 sessions of the protocol, with a search depth of 10, while the second one increased the search depth to 18 (hence it covered all possible interleavings of actions from the 3 sessions). The purpose of this was primarily to see how the verification effort scales up with the increase of search depth.

The results of the experiments are summarised in Tables 1 and 2 and Figure 4. The *Total (elapsed)* column shows the total elapsed time (wall time). The *Total (CPU)* column shows the total CPU time spent by all cores. The *GC* column shows the (elapsed) time spent on garbage collection, and *MUT* shows the actual productive time spent on the workload. The last column shows the maximum memory residency, i.e., the largest amount of memory used at any time. The two experiments look remarkably similar. We see a steep improvement in elapsed time up to 10-12 cores, before the curves flatten out. The performance speed-ups in the best cases were 5.1 (for the first experiment, with 16 cores) and 3.8 (for the second experiment, with 14 cores). In the second experiment, the single-threaded OFMC took slightly over 24 hours, but the parallel version, in the best case, terminated after 6 hours or so.

The long tails of distributions of performance gain in the previous experiments do not necessarily mean that we have reached the limit of parallelisation. Rather, it seems to be an effect on the limit of the *par-depth* (and hence also the

$\operatorname{Core} \#$	Elapse time (s)	CPU time (s)	GC time (s)	MUT (s)	Mem. res. (MB)
1	88804.862	88796.110	46791.364	42013.268	2195.3
2	56400.770	111668.513	34207.638	22193.131	6589.9
4	35479.590	133237.707	22194.304	13285.284	6766.1
6	29285.870	157276.399	18348.008	10937.858	6689.4
8	25753.490	177285.007	16184.376	9569.108	6742.4
10	24328.190	201795.566	15171.428	9156.752	6795.9
12	22916.410	221311.255	14186.897	8729.507	6823.9
14	23114.540	254643.338	14284.919	8829.614	6864.3
16	24443.140	304698.207	15330.385	9112.746	6868.6

Table 2. Kerberos protocol verification for 3 sessions and search depth 18.

memory ceiling). Tweaking the *par-depth* parameter slightly may result in better or worse performance. For the Kerberos verification, for 3 sessions and a search depth of 18, increasing the *par-depth* to 4 from 3 gained us some improvement: the elapsed time was 21765.3 seconds (so about 1.12 speed up over the execution time using a par-depth of 3), GC time 14271.5 seconds and MUT time 7494.8 seconds. However, the resident memory rose to 20GB (from 6.8GB). Raising the memory ceiling also allows more cores to be used to gain further performance improvement. For example, for the same Kerberos benchmark, with a par-depth of 4, raising the number of cores to 20 resulted in a total elapsed time of 20789.7 seconds, with resident memory of 20.8GB. Generally, increasing the par-depth results in higher memory consumption, but may allow all cores to be maximally used at all times. When tested with a par-depth of 6, the verification of Kerberos (3 sessions, search depth 18) exhausted the server's memory (196 GB) and was terminated by the operating system. These suggest that there are still performance improvements to be gained from parTreeBuffer if we can reduce the overall memory footprint of PFMC, allowing us to use a greater par-depth. However, generally, parTreeBuffer is rather unsatisfactory as the memory consumption can grow unpredictably and crash the verifier. In our next strategies, we attempt to address this issue.

Garbage collection currently seems to be the largest source of inefficiency in our experiments, taking up almost half of the execution time per core. This is, however, not due to the parallelisation, as it is also observed in the runtime for the single-core cases. Our conjecture is that this is an inherent issue with the search procedures underlying OFMC, which may involve creation of search nodes that end up not being evaluated and later discarded by the garbage collector.

4.2 Enhanced parTreeBuffer

This strategy was a modification to the original parTreeBuffer strategy which led to a dramatic improvement in both speed and memory consumption. The modification consists of two parts. First, the *par-depth* limitation is removed, and parBuffer is called at each node's children. This increases speed-up at the cost of memory overhead. However, this memory cost increase is offset by the second modification. The second change, which is the key modification, involves eagerly

```
1 parTreeBuffer :: Int \rightarrow Strategy a \rightarrow Strategy (Tree a)
2 parTreeBuffer _ strat (Node x []) = do
3 y \leftarrow strat x
4 return (Node y [])
5 parTreeBuffer c strat (Node x l) = do
6 y \leftarrow strat x
7 n \leftarrow rseq (length l)
8 l' \leftarrow parBuffer c (parTreeBuffer c strat) l
9 return (Node y l')
```

Fig. 5. parBuffer at each level, with eager evaluation of subchildren length.



Fig. 6. The effect of the depth and number of cores of the Kerberos protocol on (a) execution time and (b) memory residency, using the enhanced parTreeBuffer strategy.

evaluating the *spine* of the list of children at each node, without evaluating each individual node in the list, prior to calling **parBuffer**, as shown in Fig. 5.

To understand why the second modification is significant, recall that due to the lazy evaluation of Haskell, when a function that returns a list is called, Haskell will stop evaluating the function as soon as the topmost constructor is evaluated, i.e., when the result is of the form (x:1), where the head x and the tail 1 of the list are unevaluated expressions. In the case of PFMC, this list contains the successors of the underlying transition system encoding the protocol and the attacker moves. Deducing possible transitions may involve solving deducibility constraints, and as the number of sessions and the depth search grow, the accumulated constraints can be significant; we conjecture that evaluating this eagerly allows some of the large lazy terms to be simplified in advance, at little computation cost, which reduces the memory footprint significantly. This small change decreases the memory consumption of the strategy tenfold in many cases. Figure 6 shows the effect of the depth and number of cores on execution time and memory residency, using the basic Kerberos protocol.



Fig. 7. Conversion Ratio using the hybridSubtrees strategy for the Kerberos protocol.

4.3 parTreeChunkSubtrees

This strategy aims at controlling exactly the overall number of sparks created. It uses a concept of *fuel* and fuel splitting, motivated by [25]. It starts at the root node of the search tree with a given number of sparks (i.e., the fuel) to be created. It then divides this number between each subtree, evaluating nodes sequentially. When the number of sparks reaches one, a spark is created to evaluate the remaining subtrees in parallel. This method is extremely memory efficient and performs well on smaller trees. However, as the number of sparks requested increases, the sequential portion towards the root of the tree becomes larger. Therefore, this method is not suitable for large problem sizes. Nevertheless, the overall memory consumption of this strategy is extremely low, so it has potential applications where memory use is a priority, as well as for smaller problem sizes. An extension to this strategy which resolves this issue is hybridSubtrees.

4.4 hybridSubtrees

The hybridSubtrees strategy divides a certain number of sparks over the tree. However, instead of evaluating sequentially until it runs out of fuel, it recursively calls parBuffer at each level, and divides the remaining sparks between its children. When the strategy can no longer create a full buffer, it creates a spark for the remaining subtree. This strategy therefore has a strict upper bound on the degree of parallelism. It also affords better control over the trade-off between memory and performance in general. When given enough sparks, it behaves similarly to parTreeBuffer. Given fewer sparks, the average granularity increases, the performance slows, and the memory consumption decreases. A benefit of sparking subtrees may also be that nodes deeper in the tree are less likely to require evaluation, meaning less work for the relatively overloaded subtree sparks. This is evidenced by its conversion rate when compared to other strategies, which indicates fewer fizzled and garbage collected sparks, as shown in Fig. 7.

A limitation of this strategy is that the sparks are not evenly divided between subtrees, and the number of sparks created is usually significantly lower than the number requested. Overall, the strategy performed slightly better than parTreeBuffer at certain problem sizes. However when testing very large problem sizes (4 sessions or depths greater than 10), spark creation as a proportion of the



Fig. 8. The effect of the depth and number of cores of the Kerberos protocol on (a) execution time and (b) memory residency, using the hybridSubtrees strategy.

spark cap provided decreased, and it became difficult to assign adequate sparks to offer comparable speeds to parTreeBuffer without thoroughly testing for a given depth to explore its sparking characteristics. Fig. 8 shows the execution time and memory residency using this strategy on the Kerberos protocol.

4.5 Strategies with Annotation

The above strategies make assumptions about the shape of the tree which could impact performance for particular problems. For example, sparks could be left unused by hybridSubTrees due to the unbalanced nature of the tree. Learning more about the search tree before applying a strategy could help to offset this. We experimented with several strategies using annotation methods inspired by [25], by annotating each node in the search tree with information about the number of subnodes, in order to achieve better sparking and load balancing characteristics. The core of these strategies was at each node developing a list of slightly different strategies to apply to each subtree, based on the annotation information. For example, in the case of fuel-splitting strategies such as parTreeChunkSubtrees or hybridSubtrees, annotation was used to ensure excess fuel was not passed to subtrees without sufficient nodes to make use of it.

However, in all cases these strategies did not justify their additional cost in performance and memory. This may be due to the fact that many nodes in the tree are not evaluated under normal conditions, but the annotation runs force a degree of evaluation in order to count nodes, and this overhead is not offset by any performance gains. It is also possible that the chosen method of annotation is inefficient, and better methods using, for example, a heuristic technique may perform better.

4.6 Comparison

The enhanced parTreeBuffer strategy offers reliably good results for most problem sizes, with improved speed-up and memory consumption in all cases over



Fig. 9. (a) Best execution time vs depth and (b) average execution time for varying numbers of cores for each of the strategies.



Fig. 10. Maximum memory residency vs depth for some of the strategies.

the parTreeBuffer strategy. For small trees, the simpler parTreeChunkSubtrees performs better in both measures than any alternatives, but at these problem sizes the difference is not significant. The hybridSubTrees strategy performs well if assigned adequate fuel, but this can become difficult for greater depths.

For very large problems, where parTreeBuffer may overflow memory due to excessive spark creation, hybridSubtrees can also be used to limit parallelism while still benefiting from parallel speed-up. It performed consistently better in terms of memory consumption compared to parTreeBuffer and offers more explicit control over the degree of parallelism. Figs. 9 and 10 show the comparisons of execution time and memory for the different strategies.

These observations apply only to PFMC, and testing on a variety of other problems would be required for commenting on the more general performance characteristics of the strategies. Finer granularity of problem size at each node would make chunking strategies more applicable, whereas problems which require the evaluation of the entire tree at each execution would not benefit as much from the buffering approach and may be better served by fuel-splitting or some other strategy.

4.7 Enabling the verification of protocols with algebraic operators

OFMC also supports extensions of Dolev-Yao attacker models with algebraic operators. Some of these operators, such as XOR, are supported by default, but the user has the ability to add their own equational theory for custom operators. The custom theories, however, are supported only when the input is specified in the AVISPA Intermediate Format (IF) format [21], which is not very userfriendly. OFMC does provide a translator from AnB format to IF format, but currently there are some issues in recognising the custom theories. We have made some modifications to the translator to allow us to experiment with verification of protocols with algebraic properties. In terms of performance speed-up, the results are in line with what we have seen in the case of Dolev-Yao attackers. Further details of these experiments are available in Appendix B.

5 Conclusion and Future Work

Our preliminary results show that there is a significant improvement in moving towards moderate parallelisation of security protocol verification based on symbolic constraint solving. We managed to achieve 3-5 times speed up compared to the sequential verifier, utilising between 4-16 cores, on benchmarks with 3 sessions. Beyond 16 cores the performance improvement does not seem substantial. This is partly due to our algorithm limiting the memory usage, preventing more tasks to be executed in parallel, but in some cases it could also be that the problem has hit a limit of parallelisation, e.g., some parts of the search could not be parallelised due to inherence dependencies.

One major issue in scaling up the protocol verification is controlling the memory consumption, while attaining a good degree of parallelisation. To this end we have been experimenting with various buffering and chunking strategies. The extended parTreeBuffer and the hybridSubTree strategies (Section 4) seem to so far provide a good balance of memory consumption and performance, with parTreeBuffer generally performing better, but with a worse memory footprint.

For future work, we plan to explore annotation methods to achieve better load balancing in sparking the search trees, perhaps by moving to a limited sharedmemory concurrent setting. We also plan to apply our light-weight parallelisation approach to improve the performance of other protocol verifiers, e.g., Tamarin.

References

- M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.
- O. Almousa, S. Mödersheim, and L. Viganò. Alice and bob: Reconciling formal models and implementation. In Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday, volume 9465 of LNCS, pages 66–85. Springer, 2015.
- M. Arapinis, S. Delaune, and S. Kremer. From one session to many: Dynamic tags for security protocols. In Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008. Proceedings, volume 5330 of LNCS, pages 128–142. Springer, 2008.
- A. Armando, R. Carbone, and L. Compagna. SATMC: a sat-based model checker for security protocols, business processes, and security apis. Int. J. Softw. Tools Technol. Transf., 18(2):187–204, 2016.

- A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008.*, pages 1–10. ACM, 2008.
- M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019:1393, 2019.
- D. A. Basin, S. Mödersheim, and L. Viganò. Algebraic intruder deductions. In Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Proceedings, volume 3835 of LNCS, pages 549–564. Springer, 2005.
- B. Blanchet. Using horn clauses for analyzing security protocols. In Formal Models and Techniques for Analyzing Security Protocols, volume 5 of Cryptology and Information Security Series, pages 86–111. IOS Press, 2011.
- 9. B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1-2):1–135, 2016.
- I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. A metanotation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999*, pages 55–69. IEEE Computer Society, 1999.
- V. Cheval. APTE: an algorithm for proving trace equivalence. In Tools and Algorithms for the Construction and Analysis of Systems 20th International Conference, TACAS 2014. Proceedings, volume 8413 of LNCS, pages 587–592. Springer, 2014.
- 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, pages 529–546. IEEE Computer Society, 2018.
- V. Cortier, A. Dallon, and S. Delaune. Efficiently deciding equivalence for standard primitives and phases. In *Computer Security - 23rd European Symposium* on Research in Computer Security, ESORICS 2018, Proceedings, Part I, volume 11098 of LNCS, pages 491–511. Springer, 2018.
- V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. J. Comput. Secur., 14(1):1–43, 2006.
- D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans.* Information Theory, 29(2):198–207, 1983.
- N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In Workshop on Formal Methods and Security Protocols, 1999.
- 17. M. I. Kanovich, T. B. Kirigin, V. Nigam, and A. Scedrov. Bounded memory dolevyao adversaries in collaborative systems. *Inf. Comput.*, 238:233–261, 2014.
- S. Marlow, D. Jones, and S. Singh. threadscope (software package). https:// wiki.haskell.org/ThreadScope.
- S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011, pages 71–82. ACM, 2011.
- S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.
- 21. S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In *Foundations of Security Analysis and*

Design V, FOSAD 2007/2008/2009 Tutorial Lectures, volume 5705 of LNCS, pages 166–194. Springer, 2009.

- M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), pages 174–187. IEEE Computer Society, 2001.
- F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. J. Comput. Secur., 7(1):191–230, 1999.
- A. Tiu, N. Nguyen, and R. Horne. SPEC: an equivalence checker for security protocols. In Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Proceedings, volume 10017 of LNCS, pages 87–95, 2016.
- 25. P. Totoo. *Parallel evaluation strategies for lazy data structures in Haskell*. PhD thesis, Heriot-Watt University, 2016.
- T. van Deursen and S. Radomirovic. Attacks on RFID protocols. *IACR Cryptol.* ePrint Arch., page 310, 2008.

A Case studies

We show here more details of the tests we performed on some example protocols. These example protocols are from the original distribution of OFMC (and also included in the distribution of PFMC).

A.1 A simplified Kerberos protocol

Figure 11 shows the formalisation of a simplified version of the Kerberos protocol in the "Alice and Bob" (AnB) notation of OFMC. It abstracts away some aspects of the actual protocol, such as the authentication tags. The goal states that the protocol establishes a confidential and authentic channel between s and C. We have provided details of the performance evaluation of Kerberos in Section 4 so we will not repeat it here.

A.2 Google Single Sign-On protocol

The attack can be found within 2 sessions. The (simplified) protocol is given in Figure 12. The constructor pk denotes the public key constructor; idp is the identity provider, C is the client, S is the service provider and the private key that corresponds to a public key pk(X) is denoted by inv(pk(X)). The flaw in Google's implementation of the protocol (which was based on SAML 2.0) is in step 4 (line 20 in Figure 12): Google's implementation omitted certain information, such as the unique identifier of the authentication request (the ID variable in the protocol) and the identity of the service provider SP. We will not go into the details of the attack; the interested reader can consult [5]. This example shows a flawed Google Single Sign-On protocol (SSO); the flaw was discovered by Armando et. al. [5] using the SATMC model checker [4]. There is an attack when two sessions of the protocol are running concurrently. Thus, to prove or disprove the security goals, we set the search depth to 12 (since each session

```
1 Protocol: Basic_Kerberos # Bounded-verified
2
3 Types: Agent C,a,g,s;
         Number N1,N2,T1,T2,T3,Payload,tag;
4
         Function hash,sk;
 5
          Symmetric_key KCG,KCS
 6
 7
   Knowledge: C: C,a,g,s,sk(C,a),hash,tag;
8
        a: a,g,hash,C,sk(C,a),sk(a,g),tag;
9
        g: a,g,sk(a,g),sk(g,s),hash,tag;
        s: g,s,sk(g,s),hash,tag
11
12
13 Actions:
14
15 C \rightarrow a: C,g,N1
_{16} a \rightarrow C: {| KCG, C, T1 |}sk(a,g),
          {| KCG, N1, T1, G |}sk(C,a)
17
_{18} C \rightarrow g: {| KCG, C, T1 |}sk(a,g), {|C,T1|}KCG, s,N2
_{19} g \rightarrow C: {| KCS, C, T2 |}sk(g,s), {| KCS, N2, T2, s |}KCG
_{20} C \rightarrow s: {| KCS, C, T2 |}sk(g,s), {| C, T3 |}KCS
_{\text{21}} s \rightarrow C: {|T3|}KCS, {|tag,Payload|}KCS
22
23 Goals:
_{24} s * \rightarrow * C: Payload
```

Fig. 11. A simplified version of the Kerberos protocol

```
1 Protocol: SingleSignOn
      # the flawed version of Google's SSO from before 2008 [Armando et al.]
 2
      # in comments the standard specification (which is safe)
 3
 4
  Types: Agent C, idp, SP;
5
          Number URI, ID, Data;
 6
         Function h,sk
 7
8
9 Knowledge: C: C,idp,SP,pk(idp);
              idp: C,idp,pk(idp),inv(pk(idp));
10
              SP: idp,SP,pk(idp)
11
              where SP!=C, SP!=idp, C!=idp
13 Actions:
14
      [C] * \rightarrow * SP : C,SP,URI
15
       SP *→* [C] : C,idp,SP,ID,URI
16
17
       C *→* idp : C,idp,SP,ID,URI
18
      # google:
19
      \texttt{idp} \ \ast \rightarrow \ast \ \texttt{C}
                     : {C,idp}inv(pk(idp)),URI
20
      # standard:
21
      #idp * \rightarrow * C
                     : ID,SP,idp,{ID,C,idp,SP}inv(pk(idp)),URI
22
23
      # google:
24
      [C] * \rightarrow * SP
                    : {C,idp}inv(pk(idp)),URI
25
      # standard:
26
      #[C] * >* SP : ID, SP, idp, {ID, C, idp, SP} inv(pk(idp)), URI
27
      SP *{\rightarrow}{*} [C] : Data,ID
28
29
  Goals:
30
31
      SP authenticates C on URI
32
      C authenticates SP on Data
33
      Data secret between SP,C
34
```

Fig. 12. A flawed version of Google Single Sign-On protocol. Source: OFMC distribution.

can have at most 6 steps). For this and the next two case studies, we set the *par-depth* to 3.

We repeated the experiment 8 times, increasing the number of cores by 2 with every iteration. The result is given in Table 3. The experiment results are given in Table 3. Figure 13a shows the trend in a chart.

We note that there is a significant improvement from a single core to four cores. This trend continues to 12 cores, after which point the GC overhead seems to outweigh the performance gain. Interestingly, the GC overhead does not seem to become worse with the addition of cores; in fact, it seems to improve. However, GC accounts for almost half of the execution time. This may indicate that the sequential algorithm itself is quite inefficient. The performance gain through parallelism is offset by the significant increase of memory residency, so it would seem that availability of memory is a limiting factor in scaling up the verification. The fast growth in memory consumption is what originally motivated us to develop a buffered search strategy. As the table shows, the memory residency in this case stays relatively constant, despite the increase of the number of cores used.

Core#|Elapsed time (s)|CPU time (s)|GC time (s)|MUT (s)|Mem. res. (MB)

					,
1	85.909	85.902	41.283	44.625	4.3
2	90.450	180.114	47.981	42.462	108.2
4	66.530	254.309	37.285	29.239	122.8
6	56.280	304.819	31.705	24.568	146.3
8	48.040	328.983	26.460	21.573	131.8
10	45.480	374.103	25.568	19.907	138.0
12	38.840	364.404	20.804	18.027	143.0
14	39.930	425.004	21.485	18.439	142.6
16	43.440	524.929	24.537	18.854	147.4

Table 3. Results for the Google SSO verification for 2 sessions and search depth of 12.



Fig. 13. Execution time for (a) the SSO protocol verification and (b) the TLS protocol.

```
Protocol: TLS # Bounded-verified
1
  Types: Agent A,B,s;
3
         Number NA, NB, Sid, PA, PB, PMS;
4
         Function pk, hash, clientK, serverK, prf
6
  Knowledge: A: A,pk(A),pk(s),inv(pk(A)),{A,pk(A)}inv(pk(s)),B,hash,clientK
7
       ,serverK,prf;
          B: B,pk(B),pk(s),inv(pk(B)),{B,pk(B)}inv(pk(s)),hash,clientK,
       serverK,prf
9
  Actions:
10
      A \rightarrow B: A, NA, Sid, PA
      B \rightarrow A: NB,Sid,PB,
             {B,pk(B)}inv(pk(s))
14
      A \rightarrow B: \{A, pk(A)\}inv(pk(s)),
15
          {PMS}pk(B),
16
17
          {hash(NB,B,PMS)}inv(pk(A)),
          {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
18
             clientK(NA,NB,prf(PMS,NA,NB))
19
            {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
20
      B \rightarrow A:
             serverK(NA,NB,prf(PMS,NA,NB))
21
22
23
  Goals:
^{24}
      B authenticates A on prf(PMS,NA,NB)
25
      A authenticates B on prf(PMS,NA,NB)
26
     prf(PMS,NA,NB) secret between A,B
27
```

Fig. 14. TLS handshake protocol. Source: OFMC distribution.

For this case study, we verify the TLS handshake protocol, for 3 concurrent sessions. Figure 14 shows the formalisation of a simplified version of TLS in OFMC. Here we omit an explicit formalisation of certificates and certificate authorities. Digital signatures are also modelled using public key encryption, i.e., a digital signature is just an encryption using the private key. The various parameters in the TLS handshake protocol are also abstracted away as the (random) number PMS. The function symbol hash denotes a secure cryptographic hash function, the symbol clientK denotes the function for constructing the (symmetric) encryption key for the client, and serverK denotes the function for constructing the encryption key for the server. We do not explicitly model the MAC keys.

The TLS handshake protocol has six steps, so to verify three sessions of the protocol, we need to consider a search depth of at least 18. As in the case with SSO, we tested PFMC on this protocol with increasing numbers of cores. The results are summarised in Table 4 and Figure 15. Again we observe a similar pattern of a significant reduction in elapsed time up to around 10-12 cores, before the curve flattens. In this case however, we observe a steeper decline in total elapsed time, with around 4.4 times speed up when run on 14 cores. The GC and the MUT time are roughly the same throughout.

Core#|Elapse time (s)|CPU time (s)|GC time (s)|MUT (s)|Mem. res. (MB)

1	1318.527	1318.397	603.617	714.908	19.8
2	786.020	1555.283	393.825	392.192	1203.3
4	512.580	1894.152	269.609	242.965	1412.7
6	412.580	2153.811	218.471	194.106	1429.8
8	389.970	2555.735	206.417	183.541	1537.3
10	344.250	2692.709	180.713	163.527	1617.4
12	325.430	2925.672	168.043	157.379	1564.3
14	301.230	3085.562	155.248	145.974	1535.4
16	326.050	3722.634	168.967	157.044	1531.1
- 1'					

Table 4. Results for the TLS verification for 3 sessions and search depth of 12.



Fig. 15. Execution time TLS verification

B Enabling the Verification of Protocols with Algebraic Operators

Most approaches for protocol verification usually assume the existence of a *free algebra*. Free algebras do not include destructors. The deduction capabilities of the intruder are handled by a set of deduction rules. However, certain algebraic operators used in cryptography cannot be modelled using only deduction rules. These operators require an *equational theory*, which specifies a set of equations that describe the behaviour of the operator. An example of such an algebraic operator is the XOR (exclusive-Or) operator. The equational theory of XOR does not result in a convergent rewriting system, due to the associativity and commutativity equations.

OFMC has some limited support for handling algebraic operators. It allows users to specify a custom theory file which describes equational theories. The drawback is that support for the theory files is not complete. The theories for some standard algebraic operators are built into the code, such as XOR and encryption operators. However, using theory files to specify custom algebraic operators currently only works with models specified in the *Intermediate Format* (*IF*). There is an option for translating an AnB model into an IF model, but this still has some difficulties.

We extended PFMC to provide support for users to use custom theory files using the existing OFMC translation to IF files. Using the existing translation from AnB to IF models is not straight-forward; there are several problems to overcome in order to use the translation for checking models with custom algebraic operators.

In the new version of PFMC, users can specify the properties of several custom operators in a separate theory file and then use the translate option originally provided in OFMC to create an IF model where the custom operators are recognised. Due to the current incomplete state of OFMC's custom theory option, several manual modifications are required in order to bypass errors thrown by OFMC when performing standard checks on the model which appear to fail when the custom theory files are not considered. For example, a common error thrown is that a secret X is never known by an agent A, although A may be able to deduce the secret using the equational theory properties. After translation, the resulting IF model must then be manually edited to remove the extra modifications. In most cases, this is far less tedious and error-prone than creating an entire IF model by hand, since IF models are significantly more complex than AnB models.

The translation produced has a default number of sessions of two. Unlike AnB models, the number of sessions is fixed in IF models. In order to increase the number of sessions, separate IF files must be created for different numbers of sessions, by specifying the number required at the translation stage. The manual modifications are identical for all the files, so can simply be specified once and copied to the other files.

For several examples, the manual modifications are simple to perform, such as for the CH07 protocol from [26] and the Salary Sum and Shamir-Rivest-Adleman Three Pass protocols from [14]. Some models do not require any modifications, such as the LAK protocol described in [26]. The issue is usually in cases where the protocol contains information contained within custom operator functions, or inside nested operators, for which OFMC assumes the inner information cannot be extracted.

There are, however, certain models for which it is not straight-forward to translate them into IF models even with manual modifications. One such example is Bull's authentication protocol (described in [14]), for which it is not trivial to avoid the OFMC error checks. While in general it would be possible to prevent all the initial checks from being performed, a more robust solution for future work would be to modify the tool to use the custom theory files when translating.

B.1 Experiments

We ran several protocols which require custom algebraic theories. The execution times vs. the number of cores for the IKA, SRA Three Pass and Salary Sum protocols from [14] and the LAK and CH07 protocols from [26] are shown in Figures 16 to 19. For all of the protocols, we attempted to run them with 2, 3 and 4 sessions. However, for some cases, the verification did not terminate for over 24 hrs with higher numbers of sessions, so only lower numbers are shown here, e.g. the Salary Sum protocol could not be verified with more than 2 sessions. All of the protocols have known attacks, but for the SRA and LAK protocols, OFMC does not find any attacks (see Section B.2).

For all the models, using multiple cores produced an improvement in execution time compared to a single core. Similar to what we observed in the case of Dolev-Yao intruder models, as the number of cores increased, the execution times generally decreased. However, after around 16 cores, the execution times started to increase again, as the garbage collection overheads increased, although the overall execution time still remained below the original time for a single core.

Another interesting point to observe is that for each particular protocol, e.g. the LAK protocol in Fig. 19, the shape of the graphs are identical for different numbers of sessions, although with significantly different execution times.



Fig. 16. Execution time for the IKA protocol verification a search depth of 10 with (a) 3 sessions and (b) 4 sessions.



Fig. 17. Execution time for the SRA Three Pass protocol verification a search depth of 10 with (a) 3 sessions and (b) 4 sessions.



Fig. 18. Execution time for the CH07 protocol verification a search depth of 10 with (a) 3 sessions and (b) 4 sessions.

B.2 Attacks Not Found by OFMC

Extending OFMC to improve support for custom theories has revealed some limitations of OFMC. There are some attacks related to algebraic operators that OFMC does not find.

The LAK protocol is an example where OFMC does not find an attack which requires the properties of the algebraic operators. There is an attack on the LAK protocol where the attacker utilises the properties of XOR to obtain some secret information [26]. However, OFMC reports no attacks for this protocol. This may be due to the fact that OFMC uses a bound on the message term depth to enable the verification of algebraic operators to be decidable [7]. Unfortunately, this means that for several of the protocols we tested, OFMC could not find the known attacks.



Fig. 19. Execution time for the LAK protocol verification for a search depth of 10 with (a) 2 sessions and (b) 3 sessions

C Profiling Tamarin's parallelisation

In this section, we show our preliminary tests on Tamarin parallelisation, using selected example specifications included in the Tamarin distribution. The following experiments were performed on an Ubuntu 18.04 server, running on a machine with 192 cores Intel(R) Xeon(R) Gold 6252 and 187GB RAM. The Tamarin version we were using was version 1.7.1, from the "develop" branch of the git repository (git version: 98058d7d6282edfd087aefa97c3a3baa6a34ac63).

Table 5 shows the elapsed time for the following benchmark problems from Tamarin distribution:

```
Chen_Kudla: examples/ake/bilinear/Chen_Kudla.spthy
UM_three_pass: examples/ake/dh/UM_three_pass.spthy
commitment-protocol: examples/csf17/commitment-protocol.spthy
gcm: examples/csf19-wrapping/gcm.spthy
arpki-NoThreeUntrusted: examples/ccs14/arpki-NoThreeUntrusted.spthy
```

It seems that generally, there is little improvement in the elapsed time beyond 2 cores. Figure 20 shows some fragments of the thread profiling for the Chen_Kudla benchmark, running on 4 cores. Figure 20(a) shows an example situation where the workload was distributed evenly among the four cores. This occurred for a very short duration (less than a second). The rest of the runtime profile resembles more the situation shown in Figure 20(b) where only two cores were actively used for the actual computation (the rest was either spent on garbage collection or idle). This same pattern was observed across the benchmarks we tested (except for the arpki-NoThereUntrusted, for which we did not generate a runtime profile due to the size of the log that it generates).

For the last benchmark, we did observe a relatively more significant improvement in the elapsed time; however the gain does not seem to scale with the number of cores used. It is unclear at the moment where the bottleneck for parallelisation lies; this is left for future work.

Benchmark	1 core	2 cores	4 cores	8 cores	16 cores
Chen_Kudla	39.270s	31.380s	32.080s	32.040s	35.060s
UM_three_pass	95.310s	82.720s	84.520s	83.800s	83.260s
commitment-protocol	100.750s	73.700s	64.820s	62.080s	62.750s
gcm	89.270s	81.620s	59.280s	66.940s	78.890s
arpki-NoThreeUntrusted	6293.050s	4669.480s	3758.720s	3413.110s	3686.980s

Table 5. Iotal elabsed time	Table	5.	Total	elapsed	time
------------------------------------	-------	----	-------	---------	------



Fig. 20. Runtime profiling of Chen_Kudla benchmark