Cryptographic Protocol Verification via Supercompilation

(A Case Study)

Abdulbasit Ahmed[1], Alexei P. Lisitsa[2] and Andrei P. Nemytykh[3]

[1] Department of Computer Science, The University of Liverpool
`aahmad@csc.liv.ac.uk`

[2] Department of Computer Science, The University of Liverpool
`a.lisitsa@csc.liv.ac.uk`

[3] Program Systems Institute of Russian Academy of Sciences
`nemytykh@math.botik.ru`

Abstract

It has been known for a while [35, 36, 12] that program transformation techniques, in particular, program specialization, can be used to prove the properties of programs automatically. For example, if a program actually implements (in a given context of use) a constant function sufficiently powerful and semantics preserving program transformation may reduce the program to a syntactically trivial "constant" program, pruning unreachable branches and proving thereby the property. Viability of such an approach to verification has been demonstrated in previous works [16, 19, 18] where it was applied to the verification of parameterized cache coherence protocols and Petri Nets models [11, 20]. In this paper we further extend the method and present a case study on its application to the verification of a cryptographic protocol. The protocol is modeled by functional programs at different levels of abstraction and verification via program specialization is done by using Turchin's supercompilation method.

Keywords: Program verification, cryptographic protocols, program specialization, supercompilation, program analysis, program transformation.

1 Introduction

Progam specialization techniques traditionally have been applied for optimization purposes. For example, if in a given program $p(x, y)$ a value of the argument x is fixed to some x_0 then a specialization transformation can be applied to produce a program $q_{x_0}(y)$ such that for any value of y $p(x_0, y) = q_{x_0}(y)$. What is more, specialization exploits partial knowledge of the input and other syntactical structures of the program p to make the specialized program more efficient, e.g. by pruning the *unreachable* fragments of code.

But program transformations can also be used for analysis of programs [22] and more specifically for their verification [14, 19]. In the experiments discussed below we use a specializer based on Turchin's *supercompilation* method [36, 35, 32, 33, 19, 4] as the program specialization technique. This paper extends the authors' *parameterized testing* method (see [19]) for modeling and verification of global safety properties of parameterized protocols via supercompilation. The idea of the method is very simple and natural. In order to verify a safety property of a (parameterized) non-deterministic protocol S the later is specified as a functional program $\phi(i, \bar{x})$, where input i takes an encoding of the initial state of S and input $\bar{x}$ takes an encoding of a sequence of actions of S. The program returns a state of the protocol after executing the sequence of actions $\bar{x}$ starting in the initial state i. Let $T_P(s)$ be a *testing program* which given a state s checks whether a (safety) property P holds on s (*True* or *False*). Consider a composition $T_P(\varphi_S(i,\bar{x}))$. This program first simulates the execution of the protocol and then tests the required property. Now, provided we have used adequate encodings, the statement "the safety property P holds in any possible state reachable by the execution of the protocol S" is equivalent to the statement "the program $T_P(\varphi_S(i,\bar{x}))$ never returns the value *False*". The general method assumes that a semantics-preserving program transformation (e.g. supercompilation) is applied to $T_P(\varphi_S(i,\bar{x}))$ in order to transform it to a form from which one can easily establish the required property. For example if the statements of the form "`return False;`" have been eliminated during the transformation the required property holds. This approach has shown to be efficient for the verification of various (classes of) parameterized and infinite-state protocols and systems, including parameterized cache coherence protocols [19], Java MetaLock algorithm [17], coverablity for Petri Nets [20, 11]. The simplified theoretical model of verification via supercompilation approach is presented in [19] and the completeness of the method for the verification of coverability for Petri Nets can be found in [11, 20].

In this paper we show how to extend this approach to the verification of cryptographic protocols. We are interested in development of methods of both functional modeling of the cryptographic protocols and the capability of supercompilation (an automated program specialization method) to verify the corresponding program models.

Our case study here is a classical Needham-Schroeder Public Key authentication protocol [25]. We use the parameterized testing scheme outlined above and model the nondeterministic protocol in terms of a strict functional programming language, using an oracle guessing an evaluation path of this computing system. We supercompile the program model in a context of an initial protocol configuration and an unknown evaluation path as well as an intruder behavior. We hope for simple *syntactic* (explicit) properties of the resulting program, which allow us to conclude that the protocol satisfies a security property hidden in the semantics of the original program specification of the protocol (see above the remark on the "`return False;`" statement). Otherwise we interactively use the supercompiler (SCP4 [26, 27, 28]) for searching a possible counterexample (an attack on the protocol).

The limit on the length of this paper does not allow us to give an introduction to supercompilation. Here we should only enumerate properties of the method, which are crucial for the following discussion. The properties are: (1) a supercompiler tries to prune as many unreachable (in the context of specialization) program's branches (and more generally – formally possible evaluation's paths of the program being specialized) as it can; (2) in a sense the program is simplified in the superompilation process, so it becomes more amenable to the analysis.

The paper introducing the main original ideas of supercompilation as a program optimization method can be found in [36]: it is a paper written by the creator of supercompilation – V. F. Turchin. The most complete description of supercompilation ideas is given by V. F. Turchin in a report [34]. A simplified version of supercompilation is considered in [33]. See also [19, 27].

This paper assumes the reader has basic knowledge of concepts of functional programming, pattern matching, term rewriting systems and cryptographic protocols' specifications.

The paper is organized as follows. Sect. 2 provides an informal specification of the Needham-Schroeder public key (NSPK) protocol and the program presentation language. Sect. 3 introduces the developed basics ideas behind our functional modeling of the cryptographic protocols. In Sect. 4 these ideas are instantiated for a program model of NSPK, which is our case study. A verification attempt of the NSPK program model is discussed in Sect. 5, including of generating the classical Lowe attack on NSPK. A program model of a corrected version of NSPK successfully verified by supercompiler SCP4 is described in Sect. 6. In Sect. 7 we point out another NSPK program model verified by SCP4. Related work is discussed in Sect. 8. The residual programs generated by SCP4 have been relegated to appendices [1].

2 Preliminaries

This section deals with specification of the Needham-Schroeder public key (NSPK) protocol and the presentation language used in the protocol program model described below.

2.1 The Needham-Schroeder Public Key Protocol

Let us consider the following version of the NSPK authentication protocol [25]. NSPK involves two legal participants Alice (`A`) and Bob (`B`) who aim to authenticate each other by sending *encrypted and signed* messages via an *open* channel. The authentication is required even in the presence of an intruder. The intruder (`C`) is aware of the communication rules and tries to convince `B` that he (`C`) is `A`, observing the messages moving in the channel and sometimes replacing them. I.e. `C` may intercept a message and, if he is able to read it (e.g. after decryption), he may compose a fake message based on the read information, otherwise he may send the original message back in the channel.

We assume that all participants use public-key cryptography, that is every participant, legal or intruder, has a pair of private and public keys assigned to him/her. Everyone can use a public key of anyone else to encrypt a message, but only the holder of the corresponding private key may decrypt such an encrypted message. We denote the public keys used by the protocol participants as `EA`, `EB`, `EC` respectively.

Below we denote the clients' signatures with their names. Because the signatures are constant for a long time (they do not change from one session to another), additionally, to make the communications more secure the participants use random *unique* numbers (nonces) `rA`, `rB`, `rC` in the correspondence. The numbers depend on the sessions.

Normal (secure) NSPK evaluation (a sequence of steps/messages) maintaining of information protection is as follows:

1. `A` $\rightarrow$ `B : EB(rA,A) - A` initiates a communication session and sends (in the channel) a nonce `rA` signed with `A` and encrypted with the public key `EB`. By means of this message `A` asks for ensuring that he communicates with `B` rather than with someone else.

2. `B` $\rightarrow$ `A : EA(rA,rB) - B`, upon receiving the first message from `A`, decrypts it with his private key and in response to that sends the received nonce `rA` back to `A`. I.e. `B` confirms that he can decrypt the first message. In addition `B` signs his message with a nonce `rB` and encrypts it with the key `EA`. Now `B` asks for ensuring that he makes a contact with `A` rather than someone else. I.e. a person trying to communicate must know the private key `EA` and hence (s)he can read `B`'s reply.

3. `A` $\rightarrow$ `B : EB(rB) - A` certifies that he can read the message sent by `B`: he sends the received nonce `rB` back to `B`. This message is encrypted with `EB`.

4. After the three message exchange above participants `B` and `A` assume that they communicate with each other.

Upon completion of all the steps above the *connection between `B` and `A` is established.* The objective of the protocol session is achieved. Any other messages received by the legal participants cause concern of an unauthorized access and interrupting the session. In this case it is said about an attack on the protocol.

The NSPK specification above assumes the probability of guessing the numbers `rA`, `rB` is zero. Because an intruder may interfere in the communication between `A` and `B`, NSPK is parameterized with both an unknown number of the messages and the messages themselves. Indeed, the intruder is able to generate any number of unknown messages. Additionally, the set $\mathcal{K}$ of all public keys is also a NSPK parameter, because the concrete legal participants are unknown in advance. In the program model of the NSPK protocol below we assume that `A` may initiate an execution of the protocol by sending the first request to any participant whose public key belongs to $\mathcal{K}$. The numbers `rA`, `rB` are the parameters of a fixed communication session. Other parameters of NSPK's evolution are the messages generated by `C` - as a consequence of guessing or on the basis of analyzing the messages intercepted from the channel.

The verification objective is to prove that there exist no attacks on the protocol or otherwise to construct such an attack. What constitutes an attack here we understand as a session of the protocol has successfully compeleted but at the same time an intruder took a part in the session.[1]

Exact specification is embodied into a program model of the protocol and is presented and discussed in Section 4.

2.2 The Presentation Language

We present our program examples in a variant of a pseudocode for functional programs while the real supercompilation experiments with the programs were done in a strict (call by value) functional programming language REFAL [38, 39]. The programs given below are written as *strict* term rewriting systems based on pattern matching. The sentences in the programs are ordered from the top to the bottom to be matched. The data set is a free monoid of concatenation with an additional unary constructor, which is denoted only with its parentheses (that is without a name). The colon sign stands for the concatenation. The constant `[]` is the unit of the concatenation and may be omitted, others constants c are identifiers. The monoid of the data may be defined with the following grammar:

d `::= [] |` c `|` d_1 `:` d_2 `| (`d`)`

Thus a datum is a finite sequence (including the empty sequence). Let v, f denote a variable and a function name correspondingly, then the monoid of the corrresponding terms may be defined as follows:

t `::= [] |` c `|` v `|` f`(` $args$ `) |` t_1 `:` t_2 `| (`t`)`

$args$ `::=` t `|` t`,` $args$

To be closer to REFAL we use three kinds of variables: *s*.variables range over *identifiers* (e.g. `True`), *e*.variables range over the whole data set, while *t*.variables range over the data set excluding `[]`.

Examples of identifiers are `B2, Message, refused, Ms`. Examples of the variables are `s.rA, t.1, e.cls, e.memory, e.A_st`. I.e. the variable's names may be the identifiers or natural numbers.

3 The Principles of Modeling

This section informally explains our basic principles of modeling of the cryptographic protocols with ordered term rewriting systems based on pattern matching. The next section applies the

[1]It has turned out that this very conservative definition of tha attack does not lead to the generation of the spurious attacks on NSPK protocol. This surprising property of NSPK in the context of our modeling is interesting itself.

principles to a concrete program model.

Modeling of the dynamic of the protocols. A (parameterized) non-deterministic protocol S is specified as a functional program $\phi(i, \bar{x})$, where input i takes an encoding of the initial state of S and input $\bar{x}$ takes an encoding of a finite sequence of actions of S. More precisely, the program ϕ is a term rewriting system. Given an action x_0 and a current state of S, the program ϕ computes the following state of S and goes on to the next action. Repeating such steps ϕ returns a state of the protocol after executing the whole sequence of the actions $\bar{x}_0$ starting in the initial state i. These steps are the rewriting steps of the term rewriting system ϕ. In the case of cryptographic protocols (similar to NSPK) the state consists of the open channel's content and the protocol memory split into the memories of all participants of S. The open channel contains the only (current) message.

Privacy policy for the protocol participants' memory. The protocol memory is a sequence p_1, p_2, q_3, where p_i is the memory of the i-th legal protocol participant, while q_3 is the intruder memory. We encode this sequence with a term of the following form `Memory :  t.A : t.B : e.memory`. Here `t.A ::= (A : e.A)`, `t.B ::= (B : e.B)`, while `e.memory ::= (C:e.C) | []`. I.e. the intruder part of the memory may be omitted. The last is just a technical trick: the term `(C:e.C)` informs the intruder that the channel contains a message put by himself, while `[]` (no terms) means the channel's message was renewed by someone else.

The protocol specification requires a privacy policy for the memory of `A, B` and `C`. We achieve that by the following programming discipline. Given a participant of the protocol, the participant performs his/her *given* step of the protocol with the only term rewriting step. The patterns of the sentences, which can realize the given step, are not allowed to specify the part of the memory belonging to the other participants not taking part in this step. For example, let the active participant be `B` then the following two protocol memory patterns `Memory:t.A:(B:[]):e.memory`, `Memory:(A:e.A):(B:B2):t.C` are allowed, while the pattern `Memory:(A:to:s.EB):(B:B2):t.C` is not. The reason is as follows. The pattern `(A:e.A)` specifies the name (i.e. `A`) to whom this memory part belongs, but it does not disclose any part of the content of `A`'s memory, which is `e.A`. The patterns `t.A, t.C, e.memory` hide even the names of the corresponding participants. The pattern `(A:to:s.EB)` specifies partly `A`'s memory, but that is forbidden for `B`.

A similar programming discipline based on (lack of) knowledge of encryption keys guarantees the (in)capability of reading the encrypted messages being transmitted through the communication channel.

Intruder behavior. According with the general privacy principle described above the intruder `C` is not able to spy in the memory of the legal participants. He cannot decrypt a message if he does not know the key decrypting the message. `C` never replaces his own message in the channel. `C` is able to synthesize messages from data taken from the pattern (through its variables' values) of a sentence corresponding to `C`, but he cannot split the variables' values. For example, if the pattern includes the variables `e.xy`, `t.A`, `s.rB`, then `C` may produce the term `e.xy t.A e.xy (s.rB)`, but cannot specify (even partly) the values of these variables. See Section 4.1 for additional explanations of the intruder behavior logic.

Tracing. The program model ϕ may trace some additional information on the protocol evaluation, leaving it step by step in the program result. Such a trace may be useful for constructing an attack on the protocol, if any.

In general the actions $\bar{x}$ may be abstracted whenever the chosen abstraction together with the channel content and the memory state allow to unambiguously reconstruct the corresponding action. Additionally the discrete dynamic system ϕ may compute some information on properties of the generated states and carry it to the final result of ϕ.

4 The Program Model of NSPK

Let us consider the NSPK program model (see Figure 1). It is supposed that if the protocol started, then the open channel can contain only one message: sending a next message is replacement of the current message in the channel. The term `Key:s.EB` denotes the public key used by `A` to initiate execution of the protocol. Information on the messages sent in the channel is stored in the memory

```
Memory:(A:e.A_st):(B:e.B_st):e.C_part
```

The memory is a sequence of the protocol participants' storages. The information kept in a storage is available only to the participant corresponding to the storage. The storage for the participant `e.C_part` may be empty, in which case it is denoted by `[]`.

The function `mainNSPK` defines the input point of the protocol dynamic system. In its definition we assume that in any (correct) execution there exist at least two messages sent via the channel plus the initial message. Thus the sequence of the (abstract) messages without the initial message can be represented as the term `Ms:t.x1:t.x2:e.cls`.

The initial encrypted message sent by `A` is represented with `s.EB:(rA:A)` where the identifier `rA` represents a nonce, `A` is the name of the sender. The message is encrypted with the public key of an addressee with whom `A` would like to initiate the mutual authentication.

The function `Loop(Ms:e.cls, e.broadcast, Memory:e.memory)` models the protocol dynamic. Its first argument ranges over the finite sequences of the messages' *abstracts*. A message abstract includes its sender name and an abstract of the corresponding message (if needed). `Loop` puts a next message (`e.broadcast`) in the channel (the second argument), modifies the memory and returns a trace of the messages passed over the channel followed by a flag. The trace is a sequence of terms of the following form `(s.sender_name : e.ms_info)`. Below we use the trace for constructing an attack on NSPK. The flag is `connection` - the fact of authentication of the legal participants or `refused` - the fact of interruption of the current session. Let us consider the function sentences.

`(1):` The message sequence is empty. Mutual authentication is not established. The negotiation is interrupted.

`(2a-2b):` The first message of `B`. This fact is represented both in the first `Loop`'s argument and in the memory's part accessible to `B` which is empty `[]`. Notice that the other part of the memory is not available to `B` because the pattern `Memory:(A:e.A):(B:[]):e.memory` does disclose the content neither of `e.A` (`A`'s memory) nor `e.memory`. The `B` modifies his part of the memory, keeping in mind (recording the term `B2`) that the next his message will be the second. He traces his current message as a part of the returning result. This message is a response to the request of `A`. It confirms the fact that `B` can decrypt the message received from `A`: `B's` message contains the nonce `s.rA` sent by `A`. Additionally, the memory is cleaned from possible intruder records. Note that *this removing of the intruder storage is just a technical trick and does not lead to loss of possible attacks* (see the case `(5)` below for explanations). In the case `(2a)`, from `B`'s point of view, all the protocol rules are respected, while in the case `(2b)` `B` suspects an attack and interrupts the session.

(3a-3b): The second A's message. A checks the fact that his first message was read. That is confirmed with the nonce rA. A traces this message in the returning result. This A's response returns the nonce s.r back to the communication partner. The response is encrypted with the same public key used for the first A's message: the key was early stored in the A's memory. The other part of the memory is not available to A. Additionally, the memory is cleaned from possible intruder records (see the case (5) below). In the case (3a), from A's point of view, all the protocol rules are respected, while in the case (3b) A suspects an attack and interrupts the session.

```
mainNSPK( Ms:t.x1:t.x2:e.cls, Key:s.EB ) =
     Test( (A:s.EB:(rA:A)): Loop( Ms:t.x1:t.x2:e.cls, Message:s.EB:(rA:A),
                                             Memory:(A:to:s.EB):(B:[])));
/*1.*/
Loop( Ms, e.message, e.memory ) = refused;
/*2a.*/
Loop(Ms:(B:[]):e.cls,Message:EB:(s.rA:A),Memory:(A:e.A):(B:[]):e.memory)
     = (B:EA:(s.rA:rB)) :
       Loop( Ms:e.cls, Message:EA:(s.rA:rB), Memory:(A:e.A):(B:B2) );
/*2b.*/
Loop( Ms:(B:[]):e.cls, t.message, t.memory ) = refused;
/*3a.*/
Loop( Ms:(A:e.A):e.cls, Message:EA:(rA:s.r),
                        Memory:(A:to:s.EB):t.B:e.memory )
     = (A:s.EB:(s.r)) :
       Loop( Ms:e.cls, Message:s.EB:(s.r), Memory:(A:to:s.EB):t.B );
/*3b.*/
Loop( Ms:(A:e.A):e.cls, t.message, t.memory ) = refused;
/*4a.*/
Loop( Ms:(B:B2):e.cls, Message:EB:(rB),
             Memory:(A:e.A):(B:B2):e.memory ) = (B:end):connection;
/*4b.*/
Loop( Ms:(B:B2):e.cls, t.message,  t.memory ) = refused;
/*5.*/
Loop( Ms:(C:e.xy):e.cls, Message:EC:t.r, Memory:(A:e.A):(B:e.B) )
  = (C:e.xy): Loop(Ms:e.cls, Message:e.xy,Memory:(A:e.A):(B:e.B):(C:C1));
/*6.*/
Loop( Ms:(C:e.xy):e.cls, t.message, Memory:(A:e.A):(B:e.B):(C:C1) )
     = Loop( Ms:e.cls, t.message, Memory:(A:e.A):(B:e.B):(C:C1) );

Test( connection ) = True;
Test( e.trace : refused ) = refused;
Test( (C:e.C):e.x:connection ) = (C:e.C):e.x:False;
Test( t.x1:e.trace:connection ) = t.x1 : Test( e.trace:connection );
```

Figure 1: The NSPK program model.

(4a-4b): (4a) The second B's message: this fact is confirmed with the memory. B checks the fact that his first message was read. He does that by means of the nonce rB returned back to him. B decides that the person signed by A is the A. The authentication is established. (4b) B suspects an attack and interrupts the session.

(5): The intruder C checks in the memory that the last message sent in the channel was sent by someone else. That is the memory does not contain records written by `C`, which he enters in there and which the legal participants throw out from the memory (see above). The part of the memory belonging to the legal participants is not available to `C`. `C` knowing the open keys tries to impersonate a legal participant: by means of guessing a message `e.xy` and sending it in the channel. Here `e.xy` is an arbitrary message (*this free variable ranges over the whole data set*), therefore if `C` can decrypt the intercepted message `t.r`, then the message may include any information obtained from `t.r`. Notice that in this version of the specification we abstract away capabilities of `C` and allow him to send an arbitrary message. As a consequence, the cleaning of the intruder storage being done by the legal participants in the cases `(2)` and `(3)` does not restrict any possibility of `C` to generate messages. See further discussion of this point in the next subsection. Additionally, `C` records the term `C1` in the memory, which means that this current message is sent by himself rather than someone else. Otherwise the protocol evolution runs in an infinite loop, in which `C` should analyze the message sent by himself.

`(6):` `C` checks in the memory that the last message sent in the channel was sent by himself. If the memory contains the record `C1`, then `C` does not replace the current message in the channel: otherwise the protocol evolution passes in an infinite loop. This case is the last in the function `Loop`. That means if the message was sent by someone else, then the program goes into deadlock (pattern recognition impossible). Thus we model the NSPK evaluation deadlock (`C` cannot decrypt the intercepted message) with the program interpretation deadlock (abnormal stop).

The function `Test` checks correctness of a *fixed* NSPK evolution and if the *concrete* message sequence leads to completion of the *whole* negotiation session, in which `C` took a part, then the function returns the message sequence leading to the attack on the protocol. The third pattern of the function detects that the intruder replaced a message in the channel, and therefore the protocol is not secure.

4.1 On Parameterization of the Intruder Behavior Logic

Modeling of the behavior logic of the intruder `C` is crucial for detecting an attack on any crypto-protocol. In our program model the case (6) of the function `Loop` is technical: it only transfers the turn of reading/sending a message to the legal protocol participants. The substantial intruder behavior logic is modeled in the fifth case. Guessing a message, particularly, may include *any subtle analyses and syntheses* of the messages (including their sequence), therefore our model completely parameterizes the intruder logic. It has a potential advantage that more concrete models of intruder behaviour may fail to find an attack. Our model above, being under supercompilation, provides for completely automatic generation of the intruder logic. On the other hand, potential disadvantage of fully parameterized intruder behaviour model is that "guessing a message" `e.xy` allows generating a message, which might be not relevant at all to the protocol being considered. That may lead to generation of spurious attacks. E.g. the intruder may guess a public key, which is not accessible in principle, and may encrypt (using this key) a message and initiate an attack. Speaking formally, a protocol is not secure if not only an attack was generated, but the attack is proven to be realized.

Furthemore, that may be considered as a "*potential attack*": if the intruder might guess something, then he might break the negotiation confidentiality of the legal protocol participants. Such an attack may also be very interesting for analyzing the protocols. Below we show that the NSPK logic model chosen above does not lead to the spurious attacks. As a result of a verification attempt we will construct the only classical attack [23] on the protocol.

5 A Verification Attempt of the NSPK Program Model

The NSPK program model `P` described above terminates on any input data, returning a result or falling into an abnormal state (pattern recognition impossible). That is because the number of the messages (see the first `P`'s argument) taking part in a given session is finite (but unbounded) and is exhausted step by step. In other words, `P` is primitive recursive with respect to its first argument. This important (*syntactic*) property of `P` allows us to conclude the following. In the case the supercompiler SCP4 is able to transform `P` to a residual program `P'` such that `P'` has simple *syntactic* properties showing that `P'` is identically true on its domain, then the NSPK program model is secure.

E.g. such a syntactic property, both in REFAL terms and in the presentation language terms, is the lack of in `P`'s right-side top-level passive subexpressions without the identifier `False`. An expression is said to be top-level passive iff it does not contain any function call and it is not a part of a function call argument. If `P`' has not such a property, then that may mean both existence of an attack on NSPK and weakness of the supercompiler. In both cases the program model `P` has to be additionally analyzed.

As a result of specialization of our NSPK program model the supercompiler generates a program `P'`[2] containing two sentences with the identifier `False`. Let us consider one of them:

```
F122((B:B2):e.140, e.142, EB:(rB)) = (A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):
                                     (A:EC:(rB)):e.142:(C:EB:(rB)):(B:end):False;
```

Its right-side expression is completely passive. The second sentence has the same property. Thus we are forced to suppose that the program model is not secure or the supercompiler is not able to solve the verification task. In both cases we have to continue analyzing the program model.

5.1 Interactive Search for an Attack on NSPK

Now we start an interactive search for attacks on the program model. It is natural to consider the number of the messages sent in the channel as a working time measure of NSPK. In our model terms this number is the length of the sequence `e.cls` plus 3 (see the term `Ms:t.1:t.2:e.cls` in the input point in Figure 1 and Section 4). We will interactively use the supercompiler. We input the sequence `e.cls` with a fixed length and unknown members. When the length equals 1 or 2 the corresponding residual programs are identically true on their domains. The following input point ($ln($`e.cls`$) = 3$) given as a supercompilation task leads to the generation of an attack on NSPK:

```
                        mainNSPK( Ms:t.1:t.2:t.3:t.4:t.5, Key:s.EB )
```

The corresponding residual program can be found in Appendix A in [1]. We see the only sentence containing `False`. It is labeled with a comment. `False` stays in the completely passive right side. The residual program is a one-step program: it does not contain formal syntax loops. The sentence we are interested in is:

```
mainNSPK'( Ms:(C:EB:(rA:A)):(B:[]):(A:e.129):(C:EB:(rB)):(B:B2), Key:EC )
                               = (A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):(A:EC:(rB))
                                                            :(C:EB:(rB)):(B:end):False;
```

[2]The residual program can be found in [21].

According to the semantics of the result returned by the program model we conclude that a likely attack on NSPK is the following message sequence

```
(A:EC:(rA:A)):(C:EB:(rA:A)):(B:EA:(rA:rB)):(A:EC:(rB)):(C:EB:(rB)):(B:end)
```

We are writing a *likely* attack for two reasons: (1) the considered sentence may be unreachable during interpretation of the residual program `P'`; (2) the constructed attack may be spurious (see 4.1). The first doubt may be easily dissolved: it is a one-step program[3], hence the only input data, which may lead to the sentence, must be matched with this sentence pattern. But the pattern has no variables, therefore this input data has to be:

```
Ms:(C:EB:(rA:A)):(B:[]):(A:e.129):(C:EB:(rB)):(B:B2), Key:EC
```

We input this data both to the residual and to the source program and run both programs (by the interpreter). In such a way we make sure that both programs return the right-side expression of the sentence considered, i.e. the chosen data belongs to the domains of both the residual and the *source* program.

The second doubt above is canceled with a paper published in 1995 [23]: the constructed attack is the classical attack detected by G. Lowe. In this "man-in the-middle" attack the intruder `C` using an authentication request from a participant `A` impersonates `A` in an authentication exchange with `B`. The attack runs as follows.

1. `(A:EC:(rA:A))` - `A` initiates a session with `C`;
2. `(C:EB:(rA:A))` - `C` receives the first message, decrypts it with his key, encrypts the result with `EB` and replaces in the channel the first message with the changed one;
3. `(B:EA:(rA:rB))` - `B` taking into account that the second message signed by `A` sends the current message signed with the nonce `rB` and encoded with `EA` in the channel;
4. `(A:EC:(rB))` - `A` seeing that his previous message successfully decoded decides that `B'` key is really the key used for the first message and sends (in the channel) confirmation of reading the third message, once again using the key `EC`;
5. `(C:EB:(rB)))` - `C` intercepts and decodes the forth message, he sends the (re)encoded message ensuring `B` that the last `B'`s message was read and he is a legal protocol participant;
6. the technical `(B:end)` term just means that `B` receives the fifth message and falsely decides that `C` is `A`.

6 A Corrected Version of NSPK

Let us consider a corrected version of the protocol suggested G. Lowe [23]. The second step described above (Section 2.1) can be specified more accurately: `B` $\rightarrow$ `A : EA(rA,rB,EB)`. I.e. additionally, `B` sends his public key `EB` to `A`. As a consequence, at the third step, `A` may compare the received key with the key used for encoding his initial message. She interrupts the session if the keys do not coincide. Now the corresponding cases of the program model (the function `Loop`) must be corrected as follows.

[3]I.e. it is the only rewriting step necessary to produce the program's result.

```
...
/*2a.*/
Loop( Ms:(B:[]):e.cls, Message:EB:(s.rA:A), Memory:(A:e.A):(B:[]):e.memory )
     = (B:EA:(s.rA:rB:EB)) :
       Loop( Ms:e.cls, Message:EA:(s.rA:rB:EB), Memory:(A:e.A):(B:B2) );
...
/*3a.*/
Loop( Ms:(A:e.A):e.cls, Message:EA:(rA:s.r:s.EB),
                        Memory:(A:to:s.EB):t.B:e.memory )
     = (A:s.EB:(s.r)) :
       Loop( Ms:e.cls, Message:s.EB:(s.r), Memory:(A:to:s.EB):t.B );
...
```

The supercompilation result (see Appendix B in [1]) of the corrected program model `P` never returns `False`. Thus we conclude the model `P` is secure. The corrected version of NSPK has been successfully verified by the supercompiler SCP4.

7 Explicit Capabilities Intruder Model

Another version of the NSPK protocol specification in terms of the Refal language has been given in the MSc Dissertation of the first author [2] where the supercompiler SCP4 has been applied for finding an attack on the original protocol and for verification of the corrected version. The main difference with the version presented here is that in [2] the capabilities of the intruder have been explicitly specified within the program model. That led to much longer specification which nevertheless was sucessfully analysed by the interactive use of SCP4. Full details can be found in [2].

8 Related Work and Concluding Remarks

The history of verification of cryptographic protocols spans more than twenty years. Needham and Schroeder [25] mentioned that security protocols are prone to extremely subtle errors, and the need for techniques to verify the correctness of such protocols is great. The work of Dolev and Yao [6] was the first to address the verification of cryptographic protocols by utilizing a formal model of the protocols and the environment. Since then the various routes in the development of the verification techniques have been taken. Model checking approach, e.g. [24, 23] has been particularly successful in demonstrating the power of formal methods by discovering the flaws in the protocols by using finite state abstractions. Theorem proving, both *interactive* [30] and automated [5] allowed to approach the verification of parameterized and infinite state protocols. The technique utilizing declarative logic programming has been developed and implemented in *ProVerif* tool [3] which is capable of the efficient automated verification of large variety of cryptographic protocols. To make it possible a special modification of the standard Prolog semantics has been implemented.

Another line of work which led to the research presented in this paper is the development of the verification methods based on program transformation techniques and in particular on supercompilation.

M. Leuschel with his coauthors [14, 13] were the pioneers who suggested to apply a program specialization method for verification of various infinite state computing systems. The systems were modeled in terms of logic programs. The used method is known as partial deduction. A. Roychoudhury and C.R. Ramakrishnan in [31] have used fold/unfold transformations of logic programs for the verification of parameterized concurrent systems. F. Fioravanti, A. Pettorossi, M. Proietti and V. Senni [7, 8] proposed to use constraint logic programs, which give more powerful means for dealing with infinite sets of states. They also studied various strategies of generalization used for verification [9].

In [10] G. W. Hamilton described using of his distillation algorithm as a proof assistant for transformation of programs into a tail recursive form in which some properties of the programs can be easily verified by the application of inductive proof rules.

As mentioned above, functional modeling and verification (by supercompilation) of global *safety* properties of nondeterministic parameterized (i.e. infinite state) cache coherence protocols was studied by A. Lisitsa and A. Nemytykh [16, 19, 15, 17]. A. Lisitsa and A. Nemytykh in [20] and A. Klimov in [11] apply supercompilation to verification of Petri Nets models.

Modeling of cryptographic protocols is more subtle. The work of A. Ahmed [2] was the first to address the verification of the cryptographic protocols via supercompilation using the functional modeling of a variant of the Dolev Yao model. Antonina Nepeivoda [29] considers modeling and verifying of the ping-pong cryptographic protocols. Verification of her program models essentially uses generalization based on Turchin's relation [37].

In this paper we have presented a method for modeling of cryptographic protocols by functional programs and their exploration via program optimization.

The method was demonstrated on the Needham-Schroeder public key protocol. Using the supercompiler SCP4 we have explored the NSPK protocol and interactively detected the classical attack on the protocol. Then we have automatically verified the corrected version of the protocol. This case study provides with the guidelines to the design of a semi-decision procedure for the verification of cryptographic protocols based on supercompilation. The procedure would either terminate with the proof of the correctness of a protocol, or generate the attacks on the protocol, or in the worst case would not terminate, if the protocol is correct, but supercompiler is not powerful enough to prove that.

The paper length limit does not allow us to provide some details of the supercompilation technique and we refer the reader to [36, 35, 32, 33, 19, 4].

The fact of the successful using of the completely parameterized intruder behavior logic in the presented model reflects some properties of the NSPK protocol. It will be very interesting to describe a class of cryptographic protocols which can be verified with such an intruder model, using the presented approach without producing of spurious attacks.

The approach we advocate in this paper is very flexible and due to the use of the expressive programming language for the specification of the models and properties can cover the wide spectrum of models - from completely parameterized to more definite, such as the Dolev-Yao model and their variants [6, 2]. Exploration of various directions, their formal representation and comparison with other approaches such as ProVerif is a topic of ongoing and future work.

Acknowlegements

We are grateful to the reviewers of the paper for their generous and constructive comments, which both improved the presentation in this paper and will influence our future work.

References

[1] A. Ahmed, A. P. Lisitsa, and A. P. Nemytykh. Appendices to the paper: Cryptographic protocol verification via supercompilation (a case study). [online], 2013. Available at URL `http://www.botik.ru/pub/local/scp/refal5/NSPK-appendices.pdf`.

[2] Abdulbasit M. Ahmed. Verification of cryptographic protocols via supercompilation. Master's thesis, Department of Computer Science, University of Liverpool, 2008. 76pp, Available at URL `http://www.csc.liv.ac.uk/~alexei/A.Ahmed.dissertation.pdf`.

[3] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *In 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001. `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`.

[4] M. Bolingbroke and S. Peyton-Jones. Supercompilation by evaluation. In *the third ACM Haskell symposium on Haskell (Haskell '10)*, pages 135–146, NY, USA, 2010. ACM New York.

[5] Ernie Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[6] D. Dolev and A. C. Yao. On security of public key protocols. *IEEE trans. on Information Theory*, IT-29:198–208, 1983.

[7] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying ctl properties of infinite state systems by specializing constraint logic programs. In *the Proc. of VCL01*, volume DSSE-TR-2001-3 of *Tech. Rep.*, pages 85–96, UK, 2001. University of Southampton.

[8] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In M. Alpuente, editor, *the Proc. of LOPSTR 2010*, volume 6564 of *LNCS*, pages 164–183. Springer, 2011.

[9] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. In W. Faber and N. Leone, editors, *Theory and Practice of Logic Programming*, volume 13 / Special Issue 02 (25th GULP annual conference), pages 175–199. Cambridge University Press, March 2013.

[10] G. W. Hamilton. Distilling programs for verification. *Electronic Notes in Theoretical Computer Science*, 190(4):17–32, 2007. The Proc. of the International Conference on Compiler Optimization Meets Compiler Verification.

[11] A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *the Proc. of PSI'11*, volume 7162 of *LNCS*, pages 193–209, 2012.

[12] H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In *Proceedings of LOPSTR03*, volume 3018 of *LNCS*, pages 1–19, 2004.

[13] M. Leuschel and H. Lehmann. Coverability of reset petri nets and other wellstructured transition systems by partial deduction. In *Proc. CL 2000*, volume 1861 of *LNAI*, pages 101–115. Springer, 2000.

[14] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *LNCS*, pages 63–82, Venice, Italy, 2000.

[15] A. P. Lisitsa and A. P. Nemytykh. A note on specialization of interpreters. In *The 2-nd International Symposium on Computer Science in Russia (CSR-2007)*, volume 4649 of *LNCS*, pages 237–248, 2007.

[16] A. P. Lisitsa and A. P. Nemytykh. Verification as parameterized testing (Experiments with the SCP4 supercompiler). *Programmirovanie, (In Russian)*, 1:22–34, 2007. English translation in *J. Programming and Computer Software*, Vol. **33**, No.1, pp: 14–23, 2007.

[17] A. P. Lisitsa and A. P. Nemytykh. Experiments on verification via supercompilation. [online], 2007–2009. `http://refal.botik.ru/protocols/`.

[18] A. P. Lisitsa and A. P. Nemytykh. Extracting bugs from the failed proofs in verification via supercompilation. In Bernhard Beckert and Reiner Hahnle, editors, *Tests and Proofs: Papers Presented at the Second International Conference TAP 2008*, number 5/2008 in Reports of the Faculty of Informatics, Univesitat Koblenz-Landau, pages 49–65, April 2008.

[19] A. P. Lisitsa and A. P. Nemytykh. Reachability analisys in verification via supercompilation. *International Journal of Foundations of Computer Science*, 19(4):953–970, August 2008.

[20] A. P. Lisitsa and A. P. Nemytykh. Solving coverability problems by supercompilation. Presentation on the Workshop on Reachability Problems - RP'08, 2008.

[21] A. P. Lisitsa and A. P. Nemytykh. A fail result of an experiment on verification (via supercompilation) of a NSPK program model. [online], 2012. `http://refal.botik.ru/protocols/r_False_NSPK.ref`.

[22] A. P. Lisitsa and A. P. Nemytykh. A note on program specialization. what can syntactical properties of residual programs reveal? arXiv:1209.5407, 2012.

[23] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 1294(3):131–133, 1995.

[24] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[25] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of ACM*, 21(12):993–999, 1978.

[26] A. P. Nemytykh. The supercompiler Scp4: General structure. (extended abstract). In *the Proc. of PSI'03*, volume 2890 of *LNCS*, pages 162–170, 2003.

[27] A. P. Nemytykh. *The Supercompiler SCP4: General Structure.* URSS, Moscow, 2007. (Book in Russian).

[28] A. P. Nemytykh and V. F. Turchin. The supercompiler Scp4: Sources, on-line demonstration. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.

[29] Antonina Nepeivoda. Ping-pong protocols as prefix grammars and Turchin relation. In *Proc. of Verification and Program Transformation*, 2013.

[30] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[31] Abhik Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification of parameterized concurrent systems. In *Program Development in Computational Logic*, pages 261–290, 2004.

[32] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pages 486–479. The MIT Press, 1995.

[33] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[34] V. F. Turchin. The language Refal – the theory of compilation and metasystem analysis. Technical Report 20, Courant Institute, New York University, February 1980.

[35] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *LNCS*, pages 645–657. Springer-Verlag, 1980.

[36] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[37] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland Publishing Co., 1988.

[38] V. F. Turchin. *Refal-5, Programming Guide and Reference Manual.* New England Publishing Co., Holyoke, Massachusetts, 1989. Electronic version: `http://www.botik.ru/pub/local/scp/refal5/`, 2000.

[39] V. F. Turchin, D. V. Turchin, A. P. Konyshev, and A. P. Nemytykh. Refal-5: Sources, executable modules. [online], 2000. `http://www.botik.ru/pub/local/scp/refal5/`.