An Experiment in Ping-Pong Protocol Verification by Nondeterministic Pushdown Automata

Robert Glück
DIKU, Department of Computer Science, University of Copenhagen

An experiment is described that confirms the security of a well-studied class of cryptographic protocols (Dolev-Yao intruder model) can be verified by two-way nondeterministic pushdown automata (2NPDA). A nondeterministic pushdown program checks whether the intersection of a regular language (the protocol to verify) and a given Dyck language containing all canceling words is empty. If it is not, an intruder can reveal secret messages sent between trusted users. The verification is guaranteed to terminate in cubic time at most on a 2NPDA-simulator. The interpretive approach used in this experiment simplifies the verification, by separating the nondeterministic pushdown logic and program control, and makes it more predictable. We describe the interpretive approach and the known transformational solutions, and show they share interesting features. Also noteworthy is how abstract results from automata theory can solve practical problems by programming language means.

Keywords protocol verification, ping-pong protocols, cryptographic protocols, nondeterministic programming, two-way pushdown automata, memoizing interpreters

1 Introduction

Soon after the introduction of public-key encryption [26], it was found that an adversary can obtain a secret message sent on a network between trusted users, not by breaking the cryptographic algorithm, but by breaking the communication protocol through complex interactions with the users. A key finding by Dolev and Yao [8, 9] was that the security problem of cryptographic ping-pong protocols can be mapped onto a decidable grammar problem. They gave an algorithm for constructing, for any given ping-pong protocol, a nondeterministic finite-state automaton (regular language) representing all possible interactions between the trusted users and the adversary, and a special-purpose algorithm for deciding the security question by computing the collapsing-state relation by a closure algorithm. Their original algorithm decided the security question in time $O(n^8)$, where n is the size of the automaton [8, 9]. This was later improved to $O(n^3)$ [7]. The security of protocols is very important because public-key crypto systems are widely used for electronic communication and underpin various Internet standards.

Recently, Nepeivoda [24] showed that the security of ping-pong protocols can also be verified by program transformation. Instead of regular expressions, the protocol in question is mapped onto a prefix grammar encoded as a first-order functional program in such a way that it can be used to decide the security question by a program transformer, specifically a supercompiler. This method builds upon work solving other verification problems by general-purpose program transformation (*e.g.*, [2, 22]). The main steps of these two methods are shown on the left- and right-most branches in Fig. 1.

This paper takes another programming language approach — an interpreter for a nondeterministic language is used instead of a program transformer. We show how to write a two-way nondeterministic pushdown (2NPDA) program that searches for insecure communications in a finite-state automaton constructed by the Dolev-Yao algorithm, and interpret the program by an existing simulator for nondeterministic pushdown programs, which decides the security question in time $O(n^3)$. This approach leads to

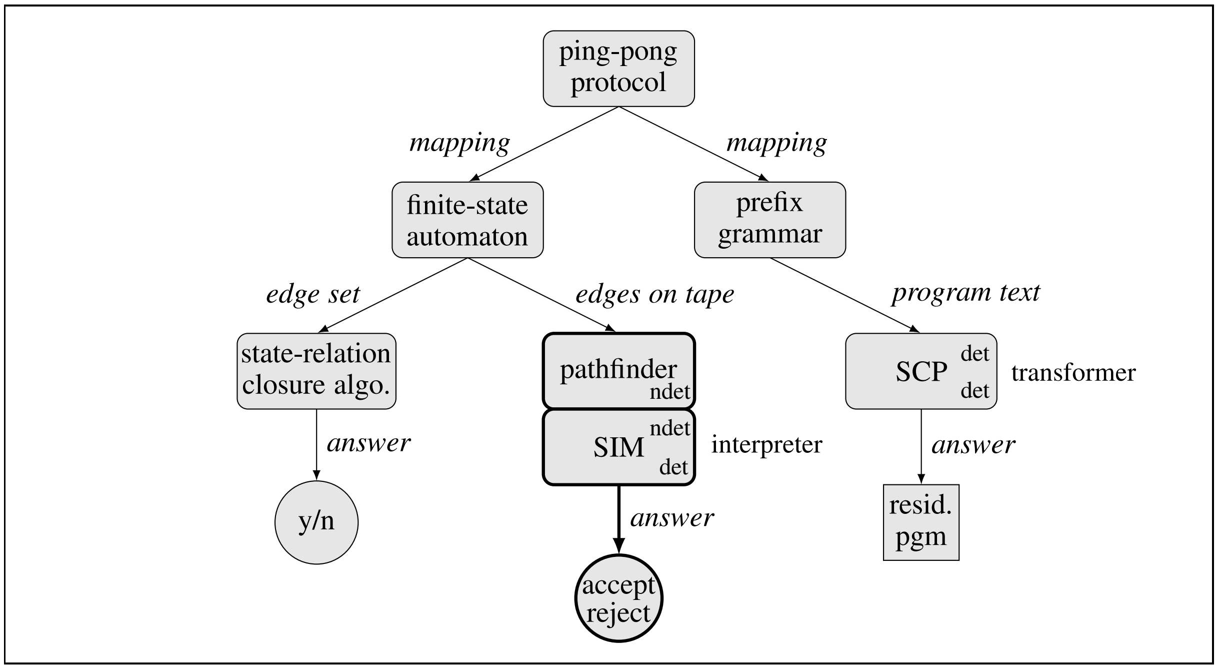

Figure 1: Three verification approaches for cryptographic ping-pong protocols: Dolev-Yao's original method [9] (left) and two programming language approaches — an interpretive method by pushdown simulation (SIM) described in this paper (in bold) and a transformative method by supercompilation (SCP) [24] (right).

a surprisingly straightforward solution that is asymptotically as efficient as any of the other verification methods above. The verification is simplified by separating the problem into an easy-to-write pushdown logic that specifies the solution in terms of finding a path in a directed graph and a control component that calculates the actual solution; such separation is known in logic programming as "algorithm = logic + control" [21]. Herein, a nondeterministic pushdown program and a simulator that embeds memoization as a control component are used. The simulator guarantees termination and polynomial-time performance on a random-access machine with a uniform cost model [3, 14]. The simulators that we use simulate one machine by another; in programming language terms these are interpreters. This experiment also shows how, relying on proven results from automata theory, a class of cryptographic protocols can be verified by means of nondeterministic programming. The method is shown by the bold shapes in Fig. 1. The deterministic (det) and nondeterministic (ndet) source and implementation languages of the simulator (SIM) and the supercompiler (SCP) are discussed in a later section. The interpretation and transformation approaches are two sides of the same coin, and we show they share interesting features.

The approach taken here is similar to other resource-bounded computation models [18] where certain properties are guaranteed for all programs regardless of how they are written (*e.g.*, all programs written in a reversible language are easily invertible [5]). Even though the nondeterministic pushdown computation model used here is subuniversal (not Turing-complete), it is not particularly weak. The multihead 2NPDA characterize the polynomial-time ("tractable") algorithms. Pushdown programming is a technique for solving problems that may deserve more attention, perhaps supported by program transformation.

In Sect. 2, we briefly review ping-pong protocols and the security problem. In Sect. 3, we introduce the nondeterministic pushdown language and, in Sect. 4, we present the protocol verifier. In Sect. 5 and 6, we discuss the methods and related work, respectively.

2 Review of Ping-Pong Protocols

Ping-pong protocols are a class of two-party cryptographic protocols. Their purpose is to transmit secret text between two users in a network. The initiator of a communication applies an initial sequence of operators to the text of a message and sends the message to the intended recipient. In each step of their communication, a participant applies an operator sequence to the text most recently received and returns the result. This ping-pong action continues several times as specified by the protocol. Operators that can be applied to a text include name stamps and cryptographic operators (see [7] for more information).

Public-key encryption is a cryptographic system that allows an *encryption key* to be revealed to the public without revealing the corresponding *decryption key* (*e.g.*, RSA [26]). Consequently, a text can be enciphered by anyone using the encryption key publicly revealed by the intended recipient of the text, but only the intended recipient can decipher the text because only this recipient has the corresponding decryption key. There is no need to secretly exchange keys between the participants.

Protocol Operators. A network is assumed to have three legitimate users (X,Y,Z) with equal rights. Each user can initiate a communication with another user by sending an initial message. A message sent in the network consists of three fields: the sender's name, the receiver's name and the text. The text is the part of a message to which a user can apply operators as specified by the communication protocol. All users have the same set of operators (Σ) that they can apply to the text of a message and each user also has a private operator (D_X,D_Y,D_Z) for decrypting a text with their private key.

Definition 1 *The operator sets of users X,Y,Z are*

$$\begin{aligned} \Sigma_X &= \Sigma \cup \{D_X\} && \textit{decrypt by private key of } X\,, && (1)\\ \Sigma_Y &= \Sigma \cup \{D_Y\} && \textit{decrypt by private key of } Y\,, && (2)\\ \Sigma_Z &= \Sigma \cup \{D_Z\} && \textit{decrypt by private key of } Z\,, && (3) \end{aligned}$$

where Σ is the common operator set available to every user:

$$\begin{aligned} \Sigma = \{ & E_X,E_Y,E_Z, && \textit{encrypt by public key of X, Y, Z} && (4)\\ & P_X,P_Y,P_Z, && \textit{prepend name of X, Y, Z}\\ & M_X,M_Y,M_Z, && \textit{match and delete prepended name of X, Y, Z}\\ & M\} && \textit{delete any prepended name}\,. \end{aligned}$$

Operators E_X,E_Y,E_Z encrypt a text with the public key of a user, operators P_X,P_Y,P_Z prepend a user name to a text, operators M_X,M_Y,M_Z delete a prepended user name if the name matches, and M deletes any prepended user name from a text. The cryptographic operators are defined for any text. If the keys mismatch, they return just gibberish (*e.g.* decryption of an encoded text with the wrong key: D_YE_X). The cryptographic operators of public-key encryption are inverse to each other (*e.g.* $D_XE_X = E_XD_X = \varepsilon$ where ε denotes the empty sequence of operators). A protocol aborts when prepended names mismatch (*e.g.* expecting another name stamp: M_YP_X). The order of applying the operators is from right to left.

Definition 2 *Cryptographic operators of public-key encryption [26] have the following identities for any user U, as have the operators for matching and deleting prepended user names [7].*

$$D_UE_U = E_UD_U = M_UP_U = MP_U = \varepsilon\,. \qquad (5)$$

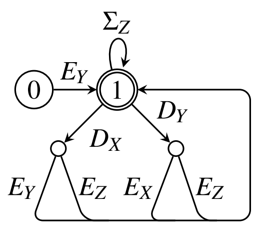

Figure 2: FSA of Protocol 1.

Consider a Dolev-Yao ping-pong protocol [7] as an example that is defined between two users $A,B \in \{X,Y,Z\}$ where A is the initiator and B the recipient. The protocol consists of two operator words, $\alpha_1 \in \Sigma_A^*$ and $\alpha_2 \in \Sigma_B^*$. In each step one of the two participants applies an operator word to the text and sends the result to the other participant. The following two words define the protocol's ping-pong action.

Protocol 1:

$$\begin{aligned} \alpha_1(A,B) &= E_B \\ \alpha_2(A,B) &= E_A D_B \end{aligned}$$

Initially, A sends a message to B after encrypting the secret text by B's public key, that is A applies $\alpha_1(A,B) = E_B$ to the text. Recipient B sends a message back to A after decrypting the received text by B's private key D_B and encrypting the result by A's public key E_A, that is B applies $\alpha_2(A,B) = E_A D_B$ to the text it received. The second step completes this ping-pong protocol. The complete operator word applied to the text is $\alpha_2(A,B) \circ \alpha_1(A,B) = E_A D_B E_B$. Under this cryptographic protocol, recipient B can read A's secret text and initiator A can compare the original text with the text echoed by B. The communication between A and B is performed without sending any plain text in the network. This cryptographic protocol appears to be a secure way of exchanging messages but, as we shall see, it is not.

Dolev-Yao Intruder Model. Assume that two of the users on the network are well-behaved (X,Y), which means they only apply the operations specified by the protocol. The third user is a saboteur (Z) who can apply any operator sequence Σ_Z^* to a message text. The saboteur waits patiently for a chance to crack the communication between the well-behaved users by listening to the network, intercepting and altering any message. This scenario is sufficient for checking the security of two-party ping-pong protocols in the Dolev-Yao intruder model.

A single saboteur is sufficient in this model because a single saboteur can do whatever a group of saboteurs can do [7]. The model assumes only a few limitations on the behavior of the saboteur. The saboteur can impersonate any sender (X,Y,Z), apply any operator sequence Σ_Z^* to a text, and send an altered message to any user. It is assumed that private keys cannot be stolen from the users.

Consider as an example how Protocol 1 can be cracked by saboteur Z. Let X initiate the communication with Y by sending a text encrypted by $\alpha_1(X,Y)$ in the network. Assume that Z intercepts X's initial message, changes the sender to Z, and sends the unchanged text to Y, who believes Z initiated a communication by sending the initial $\alpha_1(Z,Y)$. As specified by the protocol, the well-behaved user Y responds to Z by applying $\alpha_2(Z,Y)$ to the received text, that is after decrypting the text by E_X and encrypting it by E_Z for the perceived sender Z. The saboteur Z can now simply decrypt the message by D_Z. The secret has been revealed, not by cracking the public-key encryption algorithm, but the protocol! The entire operator sequence applied to the original text is reduced to ε by the operator identities:

$$D_Z \circ \alpha_2(Z,Y) \circ \alpha_1(X,Y) \;=\; \boxed{D_Z E_Z \boxed{D_X E_X}} \;=\; \varepsilon\,. \qquad (6)$$

The Security Question. The Dolev-Yao intruder model formalizes the interaction of the well-behaved users (X,Y) and the saboteur (Z) using a nondeterministic finite-state automaton (FSA). The FSA for Protocol 1 in Fig. 2 generates all operator sequences that can be applied to a text sent by initiator X. Assuming that the initiator is X is sufficient for checking the security of the protocol. The initial state is 0, the accepting state is 1, and the edges are labeled with operators. X starts by sending a message to Y that the saboteur tries to obtain by cracking the protocol. After the initial operator sequence $\alpha_1(X,Y) = E_Y$ is applied to the text, Z can intercept the message and apply to it an arbitrary operator sequence Σ_Z^*, or X and Y can apply $\alpha_2(X,Y)$, $\alpha_2(X,Z)$, $\alpha_2(Y,X)$ or $\alpha_2(Y,Z)$ to the text in response to a message received from X,Y,Z. The FSA can be constructed for any ping-pong protocol by an algorithm [7].[1]

The security question translates into the following grammar problem: A protocol is secure if the intersection of $\mathscr{L}$(FSA), the regular language defined by its FSA, and $\mathscr{L}$(G), the context-free language of all reducible operator words, is empty. Thus, the security question of ping-pong protocols is a decidable grammar problem, namely the emptiness of the intersection of a regular language and a context-language:

$$\mathscr{L}(\mathrm{FSA}) \cap \mathscr{L}(\mathrm{G}) \;\overset{?}{=}\; \emptyset\,. \qquad (7)$$

The context-free grammar G generating all reducible words is the same for all ping-pong protocols:

$$\mathrm{G} \;::=\; D_U\,\mathrm{G}\,E_U \mid E_U\,\mathrm{G}\,D_U \mid M_U\,\mathrm{G}\,P_U \mid M\,\mathrm{G}\,P_U \mid \mathrm{G}\,\mathrm{G} \mid \varepsilon \quad \text{for all } U \in \{X,Y,Z\}\,. \qquad (8)$$

The saboteur in the Dolev-Yao intruder model can apply any operator sequence Σ_Z^* to a message text, so G must generate all reducible words. A word in G is balanced with respect to "opening" and "closing" pairs of operators. Any word in $\mathscr{L}$(G) can be *reduced* to ε by repeatedly applying the identity rules in Def. 2, *i.e.* by substituting ε successively for every occurrence of pairs to which the identity rules apply. It can easily be shown that these reductions can be performed in any order without changing the result. The *reduction strategy* we are going to use later is to repeatedly apply the identity rules to the rightmost, innermost pair to which they apply.[2] There is no difference between handling a mismatch of keys and of prepended names (the operator pair does not reduce).

Definition 3 *A ping-pong protocol is* secure *iff there is no accepting path in its FSA representation whose word is reducible to ε by the operator identities; otherwise, the protocol is insecure [7].*

Protocol 1 can be made secure by prepending the name of initiator A to the text before encrypting it with B's public key, that is A applies $\alpha_1(A,B) = E_B P_A$ to the text. Recipient B now encrypts using A's public key only if the text has A's name prepended, as checked by match M_A, that is B applies $\alpha_2(A,B) = E_A M_A D_B$ to the text it received. The FSA of Protocol 2 in Fig. 3 has no accepting path whose word is reducible [7].

Protocol 2:

$$\alpha_1(A,B) \;=\; E_B P_A$$
$$\alpha_2(A,B) \;=\; E_A M_A D_B$$

Protocol 3:

$$\alpha_1(A,B) \;=\; E_B P_A E_B$$
$$\alpha_2(A,B) \;=\; E_A D_B M_A D_B$$

[1] We use the reversed and simplified version of the FSA constructed by the algorithm.

[2] $\mathscr{L}$(G) is an ambiguous language: parenthesizing is not necessarily unique, *e.g.* $\boxed{E_X \boxed{D_X E_X} D_X}$ or $\boxed{E_X D_X}\boxed{E_X D_X}$.

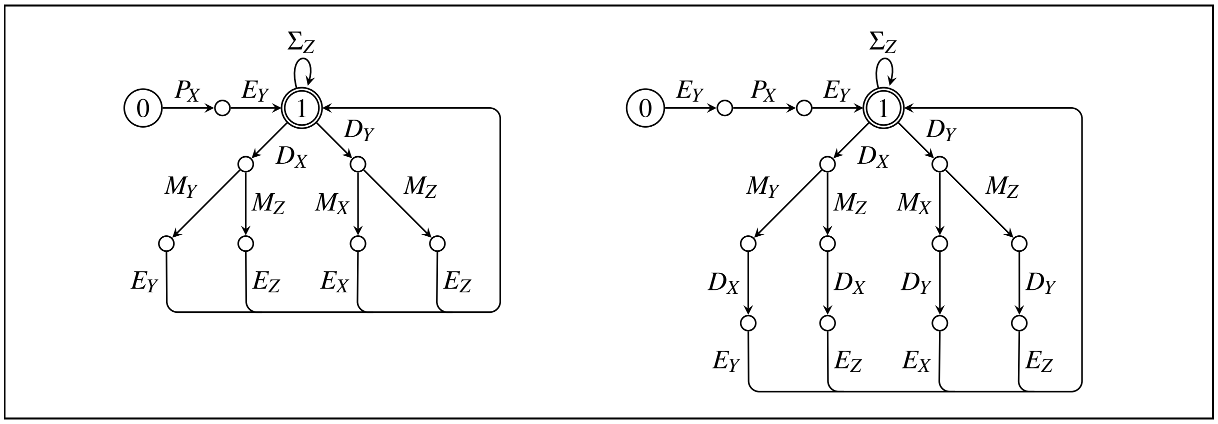

Figure 3: FSA of Protocol 2 (secure) and Protocol 3 (insecure).

One may think that extra encryption makes a protocol even more secure, but this is not necessarily the case. Suppose we want to improve Protocol 2 by encrypting the text once more. Let initiator A, before prepending the name to the text by P_A, encrypt it with B's public key, that is $\alpha_1(A,B) = E_B P_A E_B$. Receiver B now decrypts the text before encrypting it with A's public key, that is $\alpha_2(A,B) = E_A D_B M_A D_B$. This innocent looking "improvement" makes the protocol insecure! This shows that informal arguments are error prone. A formal way to decide the security of ping-pong protocols is needed. The operators which a saboteur can inject into the communication are not immediately obvious in Fig. 3 in the FSA of Protocol 3 [7]. For example, it can be cracked by a patient saboteur who twice injects operators and fakes senders and receivers:

$$\begin{aligned} D_Z \circ \alpha_2(Z,Y) \circ E_Y P_Z M_X D_Z \circ \alpha_2(Z,Y) \circ E_Y P_Z \circ \alpha_1(X,Y) &= \\ \boxed{D_Z E_Z}\boxed{D_Y \boxed{M_Z \boxed{D_Y E_Y} P_Z} \boxed{M_X \boxed{D_Z E_Z} \boxed{D_Y \boxed{M_Z \boxed{D_Y E_Y} P_Z} E_Y} P_X} E_Y} &= \varepsilon\,. \end{aligned} \tag{9}$$

3 Nondeterministic Programming and a Pushdown Language

Two-way nondeterministic pushdown automata (2NPDA) will be our programming model. Pushdown automata are simple versatile devices comprising three components: a read-only input tape, a potentially infinite stack, and a finite-state control. They can move the head on the tape in both directions, push and pop symbols to and from the stack, and test the symbol on top of the stack and the symbol read on the tape. The alphabets of tape symbols, stack symbols and states are finite. *Multihead pushdown automata* can read and move multiple heads independently on the tape.

Even though these devices are subuniversal, they are not particularly weak. The *multihead 2NPDA* are equivalent to the polynomial-time algorithms [30]. Any k-head 2NPDA can be simulated in at most $O(n^{3k})$ steps on a random-access machine with a uniform cost model where n is the length of the tape [3]. We shall see that a single head ($k = 1$) is sufficient to check the security of ping-pong protocols, which means this takes at most cubic time.

Instead of using a traditional multi-valued transition function for defining the operation of a 2NPDA, we introduce the *multihead nondeterministic pushdown language*, shown in Fig. 4, with commands to move the head (`left`, `right`), push and pop symbols (`push`, `pop`), and halt in an accepting or rejecting final state (`accept`, `reject`). Predicates can compare two symbols (`=`), test the emptiness of the stack (`bottom`) and the ends of the tape (`leftend`, `rightend`). Global variables contain the symbol currently

```
Pgm   ::=  (Label : Seq)+                       Cmd  ::=  pop     | push Sym
Seq   ::=  Cmd | Cmd ; Seq                             |  left    | left2  | ...
Test  ::=  bottom                                      |  right   | right2 | ...
        |  leftend  | leftend2  | ...                  |  choice Seq or Seq end
        |  rightend | rightend2 | ...                  |  if Test then Seq else Seq end
        |  Sym = Sym                                   |  goto Label | skip
Sym   ::=  Const | top | hd | hd2 | ...                |  accept | reject
```

Figure 4: Syntax of the multihead nondeterministic pushdown language.

on top of the stack (`top`) and read by the tape head (`hd`). The variables are updated when the stack top changes or the head moves. Similarly, additional tape heads (`hd2`, `hd3`, ...) can be moved (`left2`, `left3`, ...) and tested (`leftend2`, `leftend3`, ...). The deterministic control-flow operators are as usual (`if`, `goto`). A program consists of labeled command sequences. Execution begins at the first command of a program. The left and right ends of the tape are marked with the tape symbols > and <, respectively.

The language resembles flowchart languages except for its additional `choice` command [18] which nondeterministically executes either command sequence Seq_1 or Seq_2, that is, the next state after the choice is not uniquely determined by the current state:

$$\texttt{choice}\ Seq_1\ \texttt{or}\ Seq_2\ \texttt{end}$$

Initially, the stack is empty and the tape heads scan the left end of the tape containing the input word. An input word is *accepted* by a nondeterministic pushdown program if there exists at least one computation sequence for the program that terminates in an `accept` command. The nondeterministic operation allows the simultaneous construction of every computation sequence for a given input. The formal language accepted by a program is the set of all input words it accepts. This is the usual definition for nondeterministic pushdown automata. The semantics of the pushdown language will not be formally defined here due to lack of space and the semantics being straightforward.

A program that contains no `left` command is *one way*; one that contains no `choice` command is *deterministic*. One-head programs accept important classes of formal languages. For example, the one-way nondeterministic pushdown (1NPDA) programs accept the context-free languages.

A textbook interpretation of a pushdown program may not terminate (*e.g.*, push forever on the stack) or take exponential time before terminating. However, memoizing simulation methods ensure termination and polynomial-time performance for all pushdown programs, because the number of possible surface configurations is polynomially bounded. Thus, every pushdown program has a definite answer (accept, reject). We refer the reader to [3, 14] for a presentation of the simulation methods, and to [6, 17] for the deterministic case. Accordingly, we shall be programming in a resource-bounded and decidable nondeterministic programming language, following the approach marked bold in Fig. 1.

4 Protocol Security Checked by Nondeterministic Pushdown Programs

Combinatorial search problems may often be simply written using nondeterministic programs. Before we show how to verify the security of ping-pong protocols, we show how to find a path between two nodes in a directed graph by a pushdown program. We then extend the pathfinding program into the desired protocol verifier, discuss the nondeterministic programs and report on simulation results for the Dolev-Yao protocols. Verification by program transformation is discussed in the next section.

```
init: push '0'; right           (* push start node, pos 1st edge  *)
loop: if top = hd               (* both nodes match?              *)
      then choice               (* make a guess: traverse or skip *)
             pop; right;        (* traverse edge                  *)
             if hd = '1' then accept end;  (* path 0 -> 1 found    *)
             push hd; right     (* push next node, pos next edge  *)
           or
             2-right end;       (* skip edge                      *)
      else 2-right end;         (* mismatch: skip edge            *)
      if rightend then move-to-leftend end;  (* return to 1st edge *)
      goto loop
```

Figure 5: Finding a path in a directed graph by a nondeterministic PDA-program. The graph is given by its edges on the input tape.

4.1 Pathfinding in a Directed Graph by a Pushdown Program

Given a *directed graph* $G = (V, E)$ with nodes $u, v \in V$ and directed edges $E = \{(u_1, v_1), \ldots, (u_n, v_n)\}$, where an edge (u_i, v_i) leads from node u_i to node v_i, the task of the pushdown program is to check whether there exists a path from a source node $s \in V$ to a target node $t \in V$ in G.

Let the names of the source and target nodes be fixed as $s = 0$ and $t = 1$. For simplicity, we assume that the node names in V are included in the tape- and stack-symbol alphabets of the automaton. G can be represented by a tape of length $O(n)$ that lists all edges in E, where > and < mark the two tape ends:

$$> u_1\ v_1\ \ldots\ u_n\ v_n <$$

The nondeterministic program shown in Fig. 5 guesses a path in G from 0 to 1, if it exists. Initially, the stack is empty and the head is positioned at the left end (>) of the tape, so 0 is pushed on the empty stack by `push '0'` and the head is positioned at u_1 of the first edge $(u_1\ v_1)$ on the tape by `right` in the first line of the program. The invariant of the main loop that follows, is that the current node of the path that the program is exploring is kept on top of the stack. This node is updated by the main loop when an edge is traversed to a new node. No other nodes are pushed on the stack, so a stack of height 1 suffices for following a path in G. Furthermore, a single head suffices for scanning the edges on the tape.

The main loop moves the head `hd` over the sequence of edges on the tape, until an edge originating in the current node on top of the stack is found (`top` = `hd`). The nondeterministic choice at this point is to either traverse the edge originating in `top` or to continue the search for another edge originating in `top`. If the edge $(\texttt{top}, v)$ is traversed and $v = 1$, then a path from 0 to 1 exists and the computation halts with `accept`; otherwise, the search continues with v as the new current node. When the right end (<) of the tape is reached during the search, that is predicate `rightend` is true, the head is repositioned at the first edge by `move-to-leftend`, and the search continues with the first edge. The main loop cycles over the edges on the tape traversing or skipping edges nondeterministically. An edge takes two positions on the tape, so two right moves skip an edge (shorthand notation `2-right`).

Command `move-to-leftend` is assumedly built-in. It can be implemented by commands

```
repeat: left; if leftend then right else goto repeat end
```

The nondeterministic logic of finding a path in a directed graph is straightforward: Follow all paths starting from 0 and accept if 1 is reached. The nondeterministic program describes how to follow all paths

```
init: push '0'; right                (* push start node, pos 1st edge   *)
loop: if top = hd                    (* both nodes match?               *)
      then choice                    (* make a guess: traverse or skip  *)
             pop; right;             (* traverse edge, check identities *)
             if (hd = 'DX' ∧ top = 'EX') ∨ (hd = 'EX' ∧ top = 'DX') ∨
                (hd = 'DY' ∧ top = 'EY') ∨ (hd = 'EY' ∧ top = 'DY') ∨
                (hd = 'DZ' ∧ top = 'EZ') ∨ (hd = 'EZ' ∧ top = 'DZ') ∨
                (hd = 'MX' ∧ top = 'PX') ∨ (hd = 'M'  ∧ top = 'PX') ∨
                (hd = 'MY' ∧ top = 'PY') ∨ (hd = 'M'  ∧ top = 'PY') ∨
                (hd = 'MZ' ∧ top = 'PZ') ∨ (hd = 'M'  ∧ top = 'PZ')
             then pop                (* reduce operator pair to ε       *)
             else push hd end;       (* trace unreducible operator      *)
             right;                  (* move to next node               *)
             if hd = '1' ∧ bottom then accept end;  (* insecure path    *)
             push hd; right          (* push next node, pos next edge   *)
           or
             3-right end;            (* skip edge                       *)
      else 3-right end;              (* mismatch: skip edge             *)
      if rightend then move-to-leftend end;  (* return to 1st edge      *)
      goto loop
```

Figure 6: Protocol security checking by a nondeterministic 2-way PDA-program. The operator-labeled directed graph representing the security problem of the protocol is given on the input tape.

in a pushdown computation model without concern for efficiency and termination (*e.g.*, cycles in graph). The control is separate (in the simulator). Both of the simulation methods for nondeterministic pushdown automata [3, 14] perform a universal search in the space of nondeterministic computations (find all computation sequences), even though an existential search (halt after first accept) decides the problem. Both use a control component consisting of memoization for avoiding redundant computations. Together, the pushdown program and the simulator constitute the algorithm for finding a path in a directed graph.

4.2 Protocol Security Checking by Pathfinding

To verify ping-pong protocols, we extend the nondeterministic pathfinding program (Fig. 5) to operator-labeled graphs that represent the security problem of a ping-pong protocol (*e.g.*, Figs. 2 and 3).

Given an *operator-labeled directed graph* $G = (V, E)$ with nodes $u, v \in V$ and directed edges $E = \{(u_1, o_1, v_1), \ldots, (u_n, o_n, v_n)\}$ labeled with the operators $o_i \in \Sigma_{XYZ} = \Sigma_X \cup \Sigma_Y \cup \Sigma_Z$ of Def. 1, the task of the pushdown program is to decide whether there exists a path from a source node $s \in V$ to a target node $t \in V$ in G along which the operator word is reducible to ε by the operator identities of Def. 2.

As above, let the names of the source and target nodes be fixed as $s = 0$ and $t = 1$. Assume the node names in V and the operator names in Σ_{XYZ} are included in the tape- and stack-symbol alphabets of the automaton. G can be represented by a tape of length $O(n)$ that lists all edges in E:

$$> u_1\ o_1\ v_1\ \ldots\ u_n\ o_n\ v_n <$$

An edge is represented by three symbols on the tape. As with pathfinding, the main loop of the pushdown program cycles over the edges on the tape and guesses a path from 0 to 1, if it exists. In addition, the program collects the operators labeling the edges along the path and tries to reduce the operator word to ε by applying the operator identities. The program is shown in Fig. 6. The main parts are as follows:

- Main loop with nondeterministic choice to guess a path.
- Applying the operator identities.

The current node of the path being explored is kept on top of the stack while searching for matching edges. The operators that could not be reduced so far along the current path are on the stack below the current node. The task of the main loop is to find edges originating in the current node by cycling over the edges on the tape. An edge takes three positions on the tape, so three right moves skip an edge (shorthand notation 3-right). When an edge $(\texttt{top}, o, v)$ originating in the current node `top` is nondeterministically selected for traversal by `choice`, the current node is popped and an attempt is made to reduce the operator o labeling the selected edge and the last unreduced operator o_{top}, now on top of the stack, by trying all operator identities of Def. 2. If an operator identity applies, $o\ o_{top} = \varepsilon$, then o_{top} is popped from the stack in the then-branch of the second conditional `if` (*e.g.*, `hd = 'MX'` $\wedge$ `top = 'PX'` in the test). Otherwise, the new operator o is pushed on the stack in the else-branch because the operator pair is not reducible. This is the rightmost, innermost reduction strategy discussed in Sect. 2. Next, v is pushed and the search continues with v as the new current node, unless $v = 1$ and the stack is empty.

If a path exists from 0 to 1 along which all operators can be reduced to ε, which means the stack is empty (predicate `bottom` is true) at the target node, the program halts with `accept`. This tells us that the protocol is *insecure*. The protocol is *secure* if no path from 0 to 1 is labeled with a reducible word, that is, the input is rejected. The program defines how to search for an insecure path using a nondeterministic choice.

Tape Representation. The representation of the protocol graphs in Figs. 2 and 3 on the tape of the pushdown program is shown in Fig. 7. The node names (`0`, `1`, ...) and operator names (`EX`, `EY`, ...) are included in the tape symbols, so each takes one position on the tape. The length of the tape is $O(|E|)$ given a graph $G = (V, E)$. The representation of the 18 edges of Protocol 1 takes 56 symbols, the 23 edges of Protocol 2 take 71 symbols, and the 28 edges of Protocol 3 take 86 symbols including the two endmarkers. Names are introduced for intermediate nodes because each edge can only be labeled by a single operator. For example, the initial word of user X in Protocol 3 between source node `0` and target node `1` takes three labeled edges with two intermediate nodes, which we call `8` and `9`.

(If node names are encoded as numbers in a fixed set of tape symbols (*e.g.*, `0`, `1`), the tape length would be $O(|E| \cdot \log |V|)$, and the program would need to be adapted to deal with the encoding and separators between the edges on the tape, *e.g.*, edge $(3, E_X, 5) =$ `1 1 EX 1 0 1 $`.)

Experiments and Implementation. The pushdown program in Fig. 6 defines a 1-head, 2-way and nondeterministic pushdown automaton (1-head 2NPDA). This is easy to see because the program has a single head variable (`hd`), moves in both directions (`left`, `right`), and has a nondeterministic choice (`choice`). Because a k-head 2NPDA can be simulated in at most $O(n^{3k})$ steps where n is the length of the tape, checking the security of ping-pong protocols by a 1-head 2NPDA takes at most $O(n^3)$ steps.

The pushdown simulator [14] takes a set of transition rules as the definition of a pushdown automaton. It takes 690 transition rules to define the program shown in Fig. 6 as a 1-head 2NPDA with 8 control states, 30 tape symbols, and 30 stack symbols. The number of transition steps and surface configurations (state $\times$ stack-top symbol $\times$ tape symbol) the simulator takes to verify the three Dolev-Yao protocols

```
Protocol 1:
> 0 EY 1                                                  initial word of trusted user X (α1)
  1 EX 1 1 PX 1 1 MX 1 1 EY 1 1 PY 1 1 MY 1               words of attacker Z (Σ*Z)
  1 EZ 1 1 PZ 1 1 MZ 1 1 M  1 1 DZ 1
  1 DX 2 2 EY 1 2 EZ 1 1 DY 3 3 EX 1 3 EZ 1 <             words of trusted users X,Y (α2)

Protocol 2:
> 0 PX 8 8 EY 1                                           initial word of trusted user X (α1)
  1 EX 1 1 PX 1 1 MX 1 1 EY 1 1 PY 1 1 MY 1               words of attacker Z (Σ*Z)
  1 EZ 1 1 PZ 1 1 MZ 1 1 M  1 1 DZ 1
  1 DX 2 2 MY 5 5 EY 1 2 MZ 6 6 EZ 1                      words of trusted users X,Y (α2)
  1 DY 3 3 MX 4 4 EX 1 3 MZ 7 7 EZ 1 <

Protocol 3:
> 0 EY 8 8 PX 9 9 EY 1                                    initial word of trusted user X (α1)
  1 EX 1 1 PX 1 1 MX 1 1 EY 1 1 PY 1 1 MY 1               words of attacker Z (Σ*Z)
  1 EZ 1 1 PZ 1 1 MZ 1 1 M  1 1 DZ 1
  1 DX 2 2 MY 4 4 DX 11 11 EY 1 2 MZ 5 5 DX 12 12 EZ 1    words of trusted users X,Y (α2)
  1 DY 3 3 MX 6 6 DY 10 10 EX 1 3 MZ 7 7 DY 13 13 EZ 1 <
```

Figure 7: Tape representation of the protocol graphs in Figs. 2 and 3 (nodes and operators are represented by tape symbols `0`, `1`, ..., `EX`, `EY`, ...).

are listed in Table 1. The simulator looks for a universal solution, exploring all computation sequences leading to an accept (insecure protocol) and not halting at the first accept being found. Thus, the outcome and the order of the edges on the tape have no significant influence on the performance of the simulation.

Table 1: Simulation of the pushdown program as 2NPDA with the Dolev-Yao example protocols.

verification	*edges*	*tape*	*configs*	*steps*	*answer*
Protocol 1	18	56	584	8100	accept (insecure)
Protocol 2	23	71	740	6031	reject (secure)
Protocol 3	28	86	1184	11412	accept (insecure)

Discussion. The nondeterministic pushdown verifier is surprisingly simple (in the author's opinion), especially when considering that Dolev and Yao's first algorithm took $O(n^8)$ steps [8, 9], while the verifier in Fig. 6 is guaranteed to take at most $O(n^3)$ steps thanks to fundamental results of automata theory. The verifier checks whether the intersection of a regular language represented on the tape as FSA (the protocol to verify) and a fixed Dyck-like language containing all canceling (insecure) words is empty. Clearly, the verifier is not limited to protocol graphs constructed by the algorithm [7]. Any operator-labeled FSA can be placed on the tape and the operator identities tested in the program can be adapted easily to other identities. Thus, any security question that can be captured by an operator-labeled FSA (any regular language) intersected with a Dyck-like language induced by a fixed set of operator identities can be decided by the verifier after adaption to the specific operator identities.

The experiment also supports the proposition that programming languages can make abstract theoretical results more accessible and applicable [18, 25] (other examples are reversible programming languages [5]). The simulation approach is also amenable to optimization by program specialization, which typically reduces the interpretive overhead by an order of magnitude. However, a memoizing interpreter for a nondeterministic language may pose additional challenges to non-trivial specialization. A downside, from a programming perspective, is that the pushdown language has no convenient data structures, just a linear tape with symbols. A search for matching edges needs to cycle over all edges on the tape and to return the head from the right end to the left end.

5 Protocol Verification by Transformation and Pushdown Simulation

Verification by general-purpose program transformers is another approach that has been proposed and used successfully to check the security of cryptographic protocols by supercompilation [2, 24].

Typically, the security problem of a protocol (*e.g.*, a ping-pong protocol) is encoded as a functional program with an additional trace parameter that constrains all nondeterministic choices such that they become deterministic. Given a trace, the functional program maps an initial state into a final state which can then be tested for validity (*e.g.*, whether the state is insecure). By specializing the program with respect to a static (known) initial state and a dynamic (unknown) trace, a supercompiler will explore the control flow of all possible traces for the given initial state.[3] If the program is specialized into a residual program from which it is immediately seen that no trace can steer the residual program into an invalid final state then the original protocol is considered secure.

A difficulty with this approach is the preparation of a functional program that specializes well, which may require considerable knowledge about supercompilation. There are obstacles, namely finding the right encoding of the security problem in a universal source language and taming the power of the supercompiler with folding, generalization and other sophisticated optimizations, especially if success depends on the way the program is written. The nondeterminism inherent in the security problem is first mapped into a deterministic program and then reintroduced by specializing the deterministic program with respect to a dynamic trace, which can make the transformation hard to predict. A nondeterministic choice in the original problem does not necessarily correspond to a nondeterministic (dynamic) choice taken by the supercompiler when exploring all possible control flows in the program. The control over these choices is therefore indirect. On the other hand, case studies [2, 22] have shown that a supercompiler [29] can solve a variety of security problems, including cryptographic ping-pong protocols [24].

In our case study, the protocol verifier is a pushdown program that explores all paths in a protocol graph by nondeterministic means. The security problem is not encoded as a functional program, but given as input data (Fig. 7) to the verifier (Fig. 6). The verifier is the same for all protocols. A branching in the graph on the tape is directly modeled by a nondeterministic choice in the pushdown program. The verifier has exactly one nondeterministic choice point; all other choices in the program are deterministic (`if`). The nondeterministic `choice` is an integral part of the language semantics and gives direct control of the problem-specific nondeterminism inherent in the security problem. The verifier is written in a language for which decidability is guaranteed, which is a major advantage of this approach. To the best of the author's knowledge, this is the first study in which the pushdown programming model has been applied to ping-pong protocols, but the model's practicality for verifying a larger class of protocols has not yet been demonstrated. At least in principle, the multihead pushdown programs can decide any computationally "tractable" verification problem.

Both approaches explore all possible paths of a security problem regardless of its representation. In one approach, this is achieved by a supercompiler specializing a program representing the problem

[3]The trace program may have additional parameters depending on the particular class of protocols and the desired answer.

Table 2: Pushdown simulation and supercompilation characteristics.

method	SIM	SCP
approach	interpretation	transformation
sound answer	yes	yes
complete answer	yes	case by case
time complexity class	= polynomial	exponential?
space complexity class	$\subseteq$ polynomial	unknown
source language	subuniversal nondeterministic tail-recursive	universal deterministic recursive

and building an internal process graph, whereas in the other approach it is achieved by a simulator of pushdown programs using memoization. The nondeterministic choice is an integral part of the pushdown language semantics, while in the case of supercompilation the nondeterminism is induced into the source program by the supercompiler's non-standard "transformation semantics" [1], not the standard semantics of the source language. Neither approach shows why a protocol is insecure or how to fix it, but a step in this direction has been taken by building attack models [24].

Table 2 summarizes the two approaches. The entries in the table are for a general-purpose supercompiler [29], bearing in mind there are different supercompiler variants (*e.g.*, [22, 24, 27]). The simulator (SIM) interprets a pushdown program for a given protocol graph (sequence of edges) on the input tape and the supercompiler (SCP) transforms the program representation of the security problem into a residual program. Provided that SIM is correct, the answer (accept/reject) is sound and complete (multihead 2NPDA are decidable). Provided that SCP is correct, the generated residual program is a sound answer, but depending on the particular protocol encoding, the residual program may not always answer the security question (*e.g.*, due to overgeneralization it contains valid and invalid final states even though the protocol is secure, or due to infinite specialization no program is generated at all). Thus, completeness of the answer is marked as 'case by case' in the table. (An exception is the verification of ping-pong protocols which was shown to be decidable for all multi-party ping-pong protocols [24].)

Exponential time complexity was conjectured for ping-pong protocol verification by supercompilation [24], while the space complexity class is unknown. The time and space complexity classes of multihead 2NPDA are guaranteed [30]. The source language of SIM is a subuniversal (not Turing-complete), nondeterministic and tail-recursive pushdown language (Sect. 3), while the source language of SCP is a universal (Turing-complete), deterministic and recursive first-order functional language. The deterministic and nondeterministic property of their source languages is also indicated in Fig. 1.

6 Related Work

It has been know for several decades that nondeterministic programs are well suited for combinatorial search problems, and in many cases even easier to write than deterministic ones [11]. An obstacle is the effective control of the often exponential-time complexity of straightforward runs and nonterminating computation paths. For two-way nondeterministic pushdown automata, a polynomial-time and terminating bottom-up simulation algorithm could be given [3]. Another simulation algorithm [14] follows top-down, all reachable computation paths, as does the one for two-way deterministic pushdown automata [17]. In these algorithms, exponential time is converted into polynomial time by sharing computations. A linear-time simulation by instrumented two-way deterministic pushdown programs is given in [23]. Certain methods of model checking [4] also make use of pushdown systems. A classic case where investigations in pure theory of pushdown automata led to a practically significant algorithm is pattern matching [20]. The case study presented here appears to be one of the first to show how a nondeterministic pushdown language can be used as a decidable programming model for protocol verification.

A series of case studies examined the verification of protocols by program transformation (*e.g.*, [22]), and in particular cryptographic protocols by supercompilation (*e.g.*, [2, 24]). A few orthogonal supercompilation principles [29] were shown to solve a number of seemingly different verification problems. Related approaches for specific verification problems have been investigated (*e.g.*, [16, 19]) and comparable results can be conjectured (*e.g.*, [12]). Interpreters have been used to improve the transformation of programs (*e.g.*, [13, 15]), which is another way to factorize supercompilation-based verification. The literature on transformation-based verification is larger than the one cited here, including approaches based on unfold/fold rules (*e.g.*, [10]). The present study examined a related, but different programming language solution, namely nondeterministic programs in a decidable computation model with guaranteed resource bounds. Related principles underlying both approaches, verification by transformation and interpretation, were discussed above (Sect. 5). Extensions of logic languages with tabulation can ensure termination and optimal known complexity for queries to a large class of practical programs [28].

7 Conclusions and Further Work

This study broadens previous studies on the verification of security by program transformation in that another programming language approach, namely program interpretation, is used. We confirmed that the security of a well-studied class of cryptographic protocols can be verified by a 1-head 2NPDA. The interpretive approach used in this experiment considerably simplified the verification, by separating nondeterministic pushdown logic from control concerns, which shows again the power of a declarative style of programming.

Program transformation and interpretation are two sides of the same coin, and we identified principles that both verification approaches share (*e.g.*, explore all possible paths of a security problem). Also noteworthy is how abstract results from automata theory can be applied to practical problems when combined with a programming language approach, and that this can yield more natural and simple solutions. This study is one of the few examples in the literature where pushdown automata have been used to answer questions other than those of formal language theory. This situation is perhaps surprising because the multihead 2NPDA programming model is equivalent to the class of polynomial-time algorithms and decidable within guaranteed time and space bounds determined by the number of heads.

Though we showed how a class of cryptographic protocols can be verified by nondeterministic programming, further work is needed before a more complete picture emerges as to the practicality of the interpretive pushdown approach. The pushdown computation model is subuniversal, and thus cannot be expected to be capable of solving all verification problems in reach of a general-purpose program transformer, such as a supercompiler using sophisticated generalization techniques and capable of generating complex recursive programs as answers. On the other hand, the multihead nondeterministic pushdown model is theoretically powerful enough to decide all polynomial-time verification problems, but whether this is as straightforward as in the case of ping-pong protocols, only further investigations will show. Verification of multi-party extended protocols [24] in further studies is warranted.

From a programming perspective, the “machine-code” transition rules of classic presentations of pushdown automata are too low-level. A step towards a more user-friendly abstraction was undertaken in this paper by employing an imperative flowchart language with deterministic and nondeterministic control-flow operators. Still, the language inherits the linear input tape from automata theory. More practical data structures and languages abstractions could be considered, *e.g.* arrays and index calculations [23], tree-structured data or cons-free functional programming languages [18].

Acknowledgements. Thanks to Antonina Nepeivoda for a concise explanation of the Dolev-Yao model and to the anonymous reviewers for their constructive feedback.

References

[1] Sergei M. Abramov & Robert Glück (2000): *Combining semantics with non-standard interpreter hierarchies*. In Sanjiv Kapoor & Sanjiva Prasad, editors: *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 1974, Springer-Verlag, pp. 201–213, doi:10.1007/3-540-44450-5_16.

[2] Abdulbasit Ahmed, Alexei P. Lisitsa & Andrei P. Nemytykh (2013): *Cryptographic protocol verification via supercompilation (a case study)*. In Alexei P. Lisitsa & Andrei P. Nemytykh, editors: *Verification and Program Transformation. Proceedings*, *EPiC Series in Computing* 16, pp. 16–29, doi:10.29007/gpsh.

[3] Alfred V. Aho, John E. Hopcroft & Jeffrey D. Ullman (1968): *Time and tape complexity of pushdown automaton languages*. *Information and Control* 13(3), pp. 186–206, doi:10.1016/S0019-9958(68)91087-5.

[4] Rajeev Alur, Ahmed Bouajjani & Javier Esparza (2018): *Model checking procedural programs*. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem, editors: *Handbook of Model Checking*, Springer-Verlag. To appear.

[5] Holger B. Axelsen & Robert Glück (2011): *What do reversible programs compute?* In Martin Hofmann, editor: *Foundations of Software Science and Computation Structures. Proceedings*, LNCS 6604, Springer-Verlag, pp. 42–56, doi:10.1007/978-3-642-19805-2_4.

[6] Stephen A. Cook (1972): *Linear time simulation of deterministic two-way pushdown automata*. In Charles V. Freiman, John E. Griffith & Jack L. Rosenfeld, editors: *Information Processing 71*, North-Holland, pp. 75–80.

[7] Danny Dolev, Shimon Even & Richard M. Karp (1982): *On the security of ping-pong protocols*. *Information and Control* 55(1-3), pp. 57–68, doi:10.1016/S0019-9958(82)90401-6.

[8] Danny Dolev & Andrew C. Yao (1981): *On the security of public key protocols (extended abstract)*. In: *Foundations of Computer Science. Proceedings*, IEEE Computer Society, pp. 350–357, doi:10.1109/SFCS.1981.32.

[9] Danny Dolev & Andrew C. Yao (1983): *On the security of public key protocols*. *IEEE Transactions on Information Theory* 29(2), pp. 198–207, doi:10.1109/TIT.1983.1056650.

[10] Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2002): *Verification of sets of infinite state processes using program transformation*. In Alberto Pettorossi, editor: *Logic Based Program Synthesis and Transformation. Proceedings*, LNCS 2372, Springer-Verlag, pp. 111–128, doi:10.1007/3-540-45607-4_7.

[11] Robert W. Floyd (1967): *Nondeterministic algorithms*. *Journal of the ACM* 14(4), pp. 636–644, doi:10.1145/321420.321422.

[12] Yoshihiko Futamura, Zenjiro Konishi & Robert Glück (2002): *Program transformation system based on generalized partial computation*. *New Generation Computing* 20(1), pp. 75–99, doi:10.1007/BF03037260.

[13] Robert Glück (1994): *On the generation of specializers*. *Journal of Functional Programming* 4(4), pp. 499–514, doi:10.1017/S0956796800001167.

[14] Robert Glück (2016): *A practical simulation result for two-way pushdown automata*. In Yo-Sub Han & Kai Salomaa, editors: *Implementation and Application of Automata. Proceedings*, LNCS 9705, Springer-Verlag, pp. 113–124, doi:10.1007/978-3-319-40946-7_10.

[15] Robert Glück & Jesper Jørgensen (1994): *Generating transformers for deforestation and supercompilation*. In Baudouin Le Charlier, editor: *Static Analysis. Proceedings*, LNCS 864, Springer-Verlag, pp. 432–448, doi:10.1007/3-540-58485-4_57.

[16] Robert Glück & Michael Leuschel (2000): *Abstraction-based partial deduction for solving inverse problems: a transformational approach to software verification*. In Dines Bjørner, Manfred Broy & Alexandre V. Zamulin, editors: *Perspectives of System Informatics. Proceedings*, LNCS 1755, Springer-Verlag, pp. 93–100, doi:10.1007/3-540-46562-6_8.

[17] Neil D. Jones (1977): *A note on linear time simulation of deterministic two-way pushdown automata*. *Information Processing Letters* 6(4), pp. 110–112, doi:10.1016/0020-0190(77)90022-9.

[18] Neil D. Jones (1997): *Computability and Complexity: From a Programming Language Perspective*. Foundations of Computing, MIT Press, Cambridge, Massachusetts.

[19] Andrei V. Klimov (2012): *Solving coverability problem for monotonic counter systems by supercompilation*. In Edmund M. Clarke, Irina Virbitskaite & Andrei Voronkov, editors: *Perspectives of Systems Informatics. Proceedings*, LNCS 7162, Springer-Verlag, pp. 193–209, doi:10.1007/978-3-642-29709-0_18.

[20] Donald E. Knuth, James H. Morris & Vaughan R. Pratt (1977): *Fast pattern matching in strings*. *SIAM Journal on Computing* 6(2), pp. 323–350, doi:10.1137/0206024.

[21] Robert Kowalski (1979): *Algorithm = logic + control*. *Communications of the ACM* 22(7), pp. 424–436, doi:10.1145/359131.359136.

[22] Alexei P. Lisitsa & Andrei P. Nemytykh (2007): *Verification as a parameterized testing (experiments with the SCP4 supercompiler)*. *Programming and Computer Software* 33(1), pp. 14–23, doi:10.1134/S0361768807010033.

[23] Torben Æ. Mogensen (1994): *WORM-2DPDAs: an extension to 2DPDAs that can be simulated in linear time*. *Information Processing Letters* 52(1), pp. 15–22, doi:10.1016/0020-0190(94)90134-1.

[24] Antonina N. Nepeivoda (2016): *Ping-pong protocols as prefix grammars: modelling and verification via program transformation*. *Journal of Logical and Algebraic Methods in Programming* 85(5), pp. 782–804, doi:10.1016/j.jlamp.2016.06.001.

[25] Bernhard Reus (2016): *Limits of Computation*. Springer-Verlag, doi:10.1007/978-3-319-27889-6.

[26] Ronald L. Rivest, Adi Shamir & Leonard M. Adleman (1978): *A method for obtaining digital signatures and public-key cryptosystems*. *Communications of the ACM* 21(2), pp. 120–126, doi:10.1145/359340.359342.

[27] Morten H. Sørensen & Robert Glück (1999): *Introduction to supercompilation*. In John Hatcliff, Torben Æ. Mogensen & Peter Thiemann, editors: *Partial Evaluation. Practice and Theory*, LNCS 1706, Springer-Verlag, pp. 246–270, doi:10.1007/3-540-47018-2_10.

[28] Terrance Swift & David S. Warren (2012): *XSB: extending Prolog with tabled logic programming*. *Theory and Practice of Logic Programming* 12(1-2), pp. 157–187, doi:10.1017/S1471068411000500.

[29] Valentin F. Turchin (1986): *The concept of a supercompiler*. *ACM TOPLAS* 8(3), pp. 292–325, doi:10.1145/5956.5957.

[30] Klaus Wagner & Gerd Wechsung (1986): *Computational Complexity*. D. Reidel Publishing Company, Dordrecht, The Netherlands.