Ping-Pong Protocols as Prefix Grammars and Turchin Relation

Antonina Nepeivoda

Program Systems Institute, Pereslavl-Zalessky, Russia
a_nevod@mail.ru

Abstract

This paper describes how to verify cryptographic protocols by a general-purpose program transformation technique with unfolding. The questions of representation and analysis of the protocols as prefix rewriting grammars are discussed. In these aspects Higman and Turchin embeddings on computational paths are considered, and a refinement of Turchin's relation is presented that allows to algorithmically decide the empty word problem for prefix rewriting grammars.

1 Introduction

It is known that even in the case when algorithms of message encryption themselves are considered as completely secure some existing cryptographic protocols that use them may be insecure. Many vulnerabilities in the protocols appear due to the use of common communication channel that can be listened and analyzed by someone other than the legal participants of the interaction. Although the problem of automatic verification of such interactions is undecidable in general [1], there were described some classes of protocols for which the verification task can have a decision procedure. In particular, Dolev and Yao presented the ping-pong model of the cryptographic protocols [2]. In a ping-pong protocol a message is a single data item encrypted by a sequence of keys. Principals can apply a finite number of encryption and decryption operations to the message. Dolev, Yao and Karp showed that this protocol model can be verified in a polynomial time if an intruder can listen the communication channel between agents and participate in the interaction on every its stage [3]. Then the question appears whether these verification results can be used while modelling the protocols not only by special tools but by general-purpose program transformation tools.

A ping-pong protocol can be naturally presented as a prefix rewriting grammar. The prefix-rewriting grammars are also used as function stack abstractions in construction of loop approximations in program analysis (for example, in V. Turchin's works on supercompilation [10]). The main disctinction is that the stack operations in supercompilation are modelled by a smaller class of the prefix grammars than the ping-pong protocols. But since Turchin's statements remain provable for the wider class of prefix grammars [7], the distinction becomes insignificant.

In this paper we show how to solve the verification problem by the general purpose program transformation technique without constructing any additional tools. The only two actions to be performed by a technique are:

1. Unfolding a computational tree of a program.
2. Terminating too long computational paths in the tree to avoid infinite unfolding [1].

[1] A description of these two techniques can be found in [9].

This paper is organized as follows. First, we introduce the classical approach to ping-pong protocol verification that uses finite automata. Then we discuss different types of protocol representations as prefix rewriting grammars. After that we show what difficulties appear in the verification of protocols as prefix grammars by program transformation techniques that use unfolding together with the scattered subword relation as a path termination criterion and discuss how to avoid these difficulties by a small refinement of the criterion. Last, we describe a way to construct prefix grammars with very long minimal tracks that end with the empty word (these tracks represent attacks on the corresponding protocol).

Our contribution is the following:

1. We show that Higman condition of path termination is too weak for successful verification of ping-pong protocols (from the class introduced in [3]) in the form of the prefix grammars.

2. Basing on the Turchin relation we define a simple refinement for this condition. We prove that this refinement allows to verify any ping-pong protocol from the class being considered.

2 Ping-Pong Protocols

Consider an information exchange between several participants that is controlled by some interaction rules. Let Σ_X be a set of actions that are available to a participant X. Some actions from Σ_X, such as encryption, decryption, letter appending, etc, are elementary; these actions do not allow their decomposition and are denoted by single letters. Other actions from Σ_X are compositions of elementary actions that are unavailable to X separately. This can happen, for example, if a participant of interaction is a user of some specific cryptographic program that does not allow him/her to use its encryption algorithm without adding his/her personal information to the message to be encrypted. The composite actions from Σ_X are represented as words consisting of letters that denote corresponding elementary actions.

Consider two participants S, R of an interaction. Let us denote an initial single data item as M. Suppose S starts the interaction corresponding to a protocol by sending R the message $\alpha_1(M)$ where $\alpha_1 \in \Sigma_S^*$. R responds by $\alpha_2(\alpha_1(M))$ ($\alpha_2 \in \Sigma_R^*$) and so on until the last action α_n is reached; the message is sent there and back as a ping-pong ball. The tuple $\langle \alpha_1, \dots, \alpha_n \rangle$ is called a ping-pong protocol.

Definition 1. *A ping-pong protocol $P(S, R)$ (where S and R denote legal participants of the protocol) is a sequence $\Gamma = \langle \alpha_1, \dots, \alpha_n \rangle$ of operator words, where $\alpha_i \in \Sigma_S^*$ if i is odd and $\alpha_i \in \Sigma_R^*$ if i is even.*

Some elementary actions in a sequence $\alpha_{i+1}\alpha_i$ may partially cancel each other. For example, consider the situation when both of the principals R, S can encrypt a message by two different keys, but each principal knows only one decryption key of the corresponding two. Then $\Sigma_S = \{E_R, E_S, D_S\}$ and $\Sigma_R = \{E_R, D_R, E_S\}$, where E_X means encryption operation and D_X means the corresponding decryption action. Let the protocol be $\langle E_R, E_S D_R \rangle$. If the initial message is M then it is first transformed by S to $E_R(M)$ and then is transformed by R to $E_S(M)$. Thus the sequence $D_R E_R$ collapses to the empty word (denoted by Λ).

The cancellations satisfy the Church–Rosser property and thus can be done in an arbitrary order. So we denote them as rules $R_l \to R_r$, where R_l is a sequence of operations to be transformed and R_r is the result of the transformation. E.g. the previous cancellation rule can be denoted as $D_R E_R \to \Lambda$. Note that the cancellations must not necessarily have the form $xy \to \Lambda$; an action may be cancelled not only by a single other action but also by a sequence of several other actions.

Assume that an intruder Z can read any message x sent from S to R and vice versa and can replace any such message x by a message $\beta(x)$, where $\beta \in \Sigma_Z^*$. A protocol is insecure iff the intruder can get the initial message M.

Definition 2. *A ping-pong protocol $P(S,R) = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$ is insecure iff there is a sequence $\langle \beta_1, \beta_2, \dots, \beta_m \rangle$, $\beta_1 \beta_2 \dots \beta_m \to \Lambda$ such that $\beta_1 = \alpha_1$, and either $\exists j(\beta_i = \alpha_j)$ and $i \equiv j(mod2)$ or $\beta_i \in \Sigma_Z^*$. The sequence $\langle \beta_1, \beta_2, \dots, \beta_m \rangle$ is called an attack on the protocol.*

Note that we allow α_i to appear more than once in an attack since an intruder can initiate multiple interactions with principals.

Example 1. *Consider the following protocol $P_a = \langle ba, b^{-1} \rangle$, where $bb^{-1} \to \Lambda$ and $b^{-1}b \to \Lambda$. Let $\Sigma_Z = \{c, d\}$ such that $c = a^{-1}a^1$ and a^{-1} is the left inverse of a ($a^{-1}a \to \Lambda$ but not $aa^{-1} \to \Lambda$), and $d = ba$. Then the protocol is insecure: Z can get a when it is sent back from R to S, then send baa to R and receive aa which collapses with c.*

In the original paper [3] the following algorithm of protocol verification is introduced. First, we build a nondeterministic finite state automaton that corresponds to the protocol in the following sense.

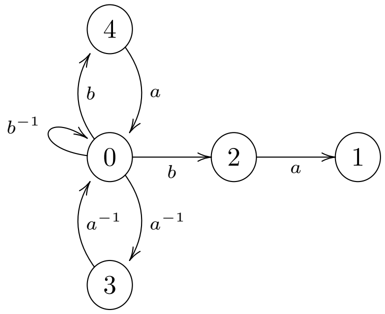
1. State 0 is the unique initial state and state 1 is the unique final state. The input alphabet is $\Sigma = \Sigma_R \cup \Sigma_S \cup \Sigma_Z$.

2. There is a directed path from 0 to 1 whose labels correspond to $\alpha_1(S,R)$.

3. For every input letter $\sigma \in \Sigma_Z$ there is a self-loop from 0 to 0, labelled σ. While we allow not only elementary actions in Σ_Z, the self-loop can contain several edges (in the original work only one-edge loops are considered).

4. For every $\alpha_i \in P$, there is a loop from 0 to 0 whose edges are labelled by the letters of α_i.

Example 2. *Let us build such an automaton for the protocol P_a.*



*Note that the action **ba** is repeated twice in the automaton.*

Let us say that a path p collapses iff its corresponding word collapses to Λ. For example, there is a collapsing path from the state (0 to the state 4) since $b^{-1}b \to \Lambda$. The set of all collapsing paths is denoted by C. To verify the protocol we must investigate whether $(0,1) \in C$. The following algorithm solves this problem [3].

1. Place in C all the pairs (i, i). Construct a queue Q which also contains all these pairs in an arbitrary order.
2. While $Q \neq \emptyset$ do
   - (a) Delete the first pair (i, j) from Q.
   - (b) If $(j, k) \in C$, $(i, k) \notin C$, then place (i, k) in C and in Q.
   - (c) If $(k, i) \in C$, $(k, j) \notin C$, then place (k, j) in C and in Q.
   - (d) If there is an edge $k \to i$ labelled τ and there is an edge $j \to l$ labelled σ, and $\tau\sigma \to \Lambda$ and $(k, l) \notin C$, then place (k, l) in C and in Q.

This algorithm terminates. The final C contains $(0, 1)$ if and only if there is a collapsing path from 0 to 1. But it can be noticed that the described type of automata actually performs stack operations, since a finite number of first symbols in the corresponding word is changed in every loop from 0 to 0. So the automata can be rewritten as grammars of the special kind and it becomes unnecessary to use special algorithms of analysis since there is a wide range of transformation and analysis tools for this kind of the grammars.

Moreover, this automata algorithm has one restriction that naturally disappears when the transition to grammars is made. The restriction is implied from the action `2(d)` of the algorithm. Since all the edges are marked with elementary actions, only cancellations of the form $xy \to \Lambda$ are processed correctly. Thus if we rewrite the conditions $c = a^{-1}a^{-1}$, $a^{-1}a \to \Lambda$ from the Example 1 as $caa \to \Lambda$ (and replace the loop from the state `0` to the state `3` and back to `0` by a self-loop marked c in the corresponding automaton) the algorithm cannot find the attack on P_a. Thus, by the transition to grammar form we become able not only to test a general-purpose program transformation technique on the verification task but also to expand the class of protocols to be verified.

Let us denote letters of an alphabet Σ by the small Latin letters $a, b, c, \ldots, p, q, r$ and the capital Latin letters A, B, C, D, E, F (maybe with subscripts or superscripts), variables that can take some value from Σ as x, y, z, w, and let us denote words from Σ^* by the Greek capitals $\Gamma, \Delta, \Phi, \Psi, \Theta$.

Definition 3. *Consider a tuple $\langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$, where Σ is an alphabet, $\Gamma_0 \in \Sigma^+$ is an initial word and $\mathbf{R} \subset \Sigma^* \to \Sigma^*$ is a set of rewrite rules. If the rewrite rules are applied only to word prefixes* $\dfrac{R : \Phi \longrightarrow \Psi}{\Phi\Theta \xrightarrow{R} \Psi\Theta}$ *then the tuple $\langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$ is a prefix rewriting grammar.*

We call a trace of a prefix rewriting grammar $\mathbf{G} = \langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$ a sequence $\{\Phi_i\}$ (finite or infinite), s.t. $\Phi_1 = \Gamma_0$ and $\forall i (i < n \Rightarrow \exists R (R : R_l \to R_r \ \& \ R \in \mathbf{R} \ \& \ \Phi_i = R_l\Theta \ \& \ \Phi_{i+1} = R_r\Theta)$.

Example 3. *Consider the "double protection" protocol P_{RR} from the paper [3] with the following modification. Let $a = E_R$, $b = E_S$, $c = E_Z$, $A = i_R$, $B = i_S$, $C = i_Z$, and i_X be the operation appending the name of X to the message. The protocol is $P_{RR} = \langle aBa, b \rangle$. Let $\Sigma_Z = \{a, c, C, c^{-1}, B^{-1}, C^{-1}\}$ and suppose that $x^{-1}x \to \Lambda$ but not $xx^{-1} \to \Lambda$ for every encryption or appending operation x. Thus, only left inverse elements are available.*

The actions $aBa \to b$ (the legal interaction between principals), $B^{-1}B \to \Lambda$ (removing the name of the agent S), $\Lambda \to c$ (encryption by the intruder's key) can be considered as rules of a prefix rewriting grammar.

There exists a hierarchy of prefix rewriting grammars (called Caucal hierarchy) that classifies the grammars by the types of their rewriting rules [5]. The Caucal hierarchy of the prefix rewriting grammars is presented in the following table [2].

Type of a grammar	Form of rewriting rules
Type 0	$\Phi \to \Psi$
Type $1\frac{1}{2}$	$pa \to q\Psi$
Type 2	$a \to \Psi$
Type 3	$a \to b \vee a \to \Lambda$

The grammars of the type 2 (and 3) are called alphabetic prefix rewriting grammars in the original Caucal work since their rules can transform only the first letter of a word.

The class $1\frac{1}{2}$ is equivalent to the class 0 (so that if some set of words can be generated by a 0-class grammar then there exists a $1\frac{1}{2}$-class grammar that generates exactly the same set of words) and wider than the class 2. The class 2 is wider than the class 3.

The ping-pong protocols are to be represented as 0- or $1\frac{1}{2}$- prefix grammars. But if we try to use straightforward representation of a ping-pong protocol as a 0-type grammar, some uncertainties can appear.

Example 4. *On the first look, the automaton for the protocol P_a (from the Example 1) can be rewritten as a* 0*-type prefix grammar as follows.*

$\mathbf{G_{PA}}$*:*

$R_1 : \Lambda \to a^{-1}a^{-1}$

$R_2 : a^{-1}a \to \Lambda$

$R_3 : bb^{-1} \to \Lambda$

$R_4 : b^{-1}b \to \Lambda$

$R_5 : \Lambda \to ba$

$R_6 : \Lambda \to b^{-1}$

Λ in the left-hand side of a rule means that the rule can be applied to any word.

In the terms of G_{PA} the question of verification is to find out whether a trace starting from the initial word ba *and ending by Λ exists. These traces represent attacks on the corresponding protocol; it is unnecessary to construct them all since the existence of even one means that the protocol is insecure.*

Despite the fact that the grammar G_{PA} looks rather simple, it is incorrect and generates some non-collapsing words that belong to collapsing paths in the automaton. For example the word $a^{-1}a^{-1}aa$ is never transformed to Λ because this demands to transform the infix $a^{-1}a$ before the first letter a^{-1}, and such actions are forbidden in prefix rewriting grammars.

The well-known way to avoid these difficulties is to use $1\frac{1}{2}$-type prefix rewriting grammars [4]. For every rule $R : pa \to q\Psi$ the letters p and q can appear nowhere but on the first position of a word. The set of these letters represents a set of control states in the corresponding automaton. Then it is possible to force all erasings to be done immediately when they can be done.

Example 5. *The $1\frac{1}{2}$-prefix grammar for the protocol P_A can look as follows ($[N]$ is a single symbol representing a state; the square brackets are introduced for better readability).*

$\mathbf{G_{PA'}}$*:*

$R_1 : [0]x \to [0]bax$

$R_2 : [0]x \to [0]a^{-1}a^{-1}x$

$R_3 : [0]x \to [0]b^{-1}x$

$R_4 : [0]b \to [0]$

$R_5 : [0]a \to [1]$

$R_6 : [1]x \to [0]a^{-1}x$

$R_7 : [1]a \to [0]$

[2] The ”negative” part of the hierarchy (types −2 and −1) is dropped since it is unnecessary for our protocol model.

x denotes an arbitrary letter from Σ. This grammar is non-deterministic and also can generate non-collapsing words that correspond to the empty word (for example, after applying R_2 to the word a)[3]*. But if there is an attack on the protocol P_A, then the grammar generates some trace ending by Λ that represents this attack.*

The representation as $1\frac{1}{2}$-type grammars is especially helpful when used together with the automaton model. But when it is used together with tree unfolding techniques it can demand some additional work, such as constructing a control state alphabet and transforming long rewriting rules in the $1\frac{1}{2}$-form. So in our investigations we use the equivalent 0-form of the grammars.

3 Ping-Pong Protocols as Prefix Grammars

Consider a ping-pong protocol as a prefix grammar in the following sense. Let every elementary action be a letter in the initial alphabet of the prefix grammar. Every action $a_1a_2a_3\dots a_n$ corresponding to an iteration from the state 0 to itself produces the following rewriting rules

$$\begin{array}{lcl} \Lambda & \to & a_1a_2\dots a_n \\ a_n^{-1} & \to & a_1a_2\dots a_{n-1} \\ a_n^{-1}a_{n-1}^{-1}\dots a_1^{-1} & \to & \Lambda \end{array}$$

In this interpretation all collapsing rules (of the form $a_ia_i^{-1} \to \Lambda$ or $a_i^{-1}a_i \to \Lambda$) can be applied immediately. This does not change properties of the protocol. The prefix grammar is finite (every loop from the state 0 to itself of the length k produces not more than $k+1$ rules). In fact, this grammar repeats the corresponding $1\frac{1}{2}$-type prefix grammar presented in the previous section but does not introduce an auxiliary state alphabet.

Example 6. *The automaton for the protocol P_a is represented by the following 0-type grammar of this sort.*

$$\begin{array}{ll} \mathbf{G_{PA0}}: & \\ R_1 : \Lambda \to ba & R_4 : b \to \Lambda \\ R_2 : \Lambda \to a^{-1}a^{-1} & R_5 : a \to a^{-1} \\ R_3 : \Lambda \to b^{-1} & R_6 : aa \to \Lambda \end{array}$$

This grammar allows doing all cancellations as early as possible. It is nondeterministic as the grammar $\mathbf{G_{PA'}}$ from 5 but contains one rule less because arbitrary left-hand sides are allowed and the rules R_5 and R_6 do not need to be decomposed to the combinations of two rules.

Consider an algorithm unfolding all possible traces of a given prefix grammar $\mathbf{G}$. This algorithm finds all the attacks on the corresponding protocol if they exist but almost all traces are infinite so the algorithm does not terminate. In general-purpose program transformation techniques (for example, supercompilation), that perform such unfoldings, some conditions of when to terminate are introduced to force termination. These conditions are not perfect because they are not specified to prefix rewriting systems and can force early terminations of finite branches (considering them as infinite). In the context of our problem this means that

[3] To avoid such cases we must introduce restrictions on x in the rules R_2, R_3, R_6.

some collapsing path in an automaton may not be found, and the task of verification may not be solved soundly by the technique. So the question raises if the existing termination conditions in, e.g. supercompilation, fit the verification task of ping-pong protocols in the form of prefix rewriting grammars, or they need some refinement.

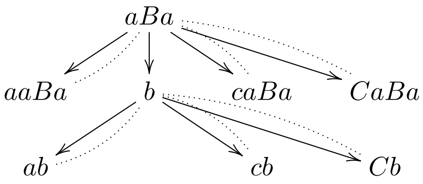
One of the popular conditions of path termination is the Higman–Kruskal embedding on terms [8]. Now we only define the Higman relation since it operates with words, as the ping-pong protocols do.

Definition 4. *Given two words in an alphabet Σ, $\Phi = a_1 a_2 \dots a_m$, $\Psi = b_1 b_2 \dots b_n$, Φ is embedded in Ψ in the sense of Higman relation ($\Phi \trianglelefteq \Psi$) iff Φ is a subsequence of Ψ. This relation is also called a scattered subword relation.*

Example 7. *Consider the "double protection" protocol P_{RR} from Example 3. Then the prefix grammar for P_{RR} can look as follows.*

$$\begin{array}{lll}\mathbf{G_{RR}}\text{:} & & \\ R_1 : \Lambda \to a & R_4 : aBa \to b & R_7 : B \to \Lambda \\ R_2 : \Lambda \to c & R_5 : aCa \to c & R_8 : C \to \Lambda \\ R_3 : \Lambda \to C & R_6 : c \to \Lambda & \end{array}$$

Let the initial word Γ_0 be aBa. If we start unfolding until finding a first such Γ and Δ that $\Gamma \trianglelefteq \Delta$, then the graph for the grammar can look as follows:



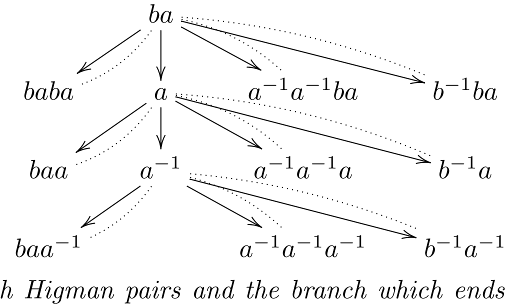
The dot arcs denote the embeddings over $\trianglelefteq$.

No path in this graph reaches Λ. But such path in the whole tree exists: trace that corresponds to it is

$$aBa \to CaBa \to aCaBa \to cBa \to Ba \to a \to Ca \to aCa \to c \to \Lambda.$$

This path corresponds to the attack on the protocol P_{RR}.

Example 8. *Let us unfold all traces of $\mathbf{G_{PA0}}$ (Example 6) until a pair Γ, Δ, such that Δ is a descendant of Γ and $\Gamma \trianglelefteq \Delta$, emerges on a branch.*



All the paths end with Higman pairs and the branch which ends with Λ is lost. Note that if the rule $aa^{-1} \to \Lambda$ is allowed then such branch (but not all possible such branches) is found even by the unfolding with the Higman condition.

Therefore Higman relation itself is not enough as a termination condition while working with prefix rewriting grammars that are generated from protocols. In the next section we consider some other classical termination condition that reveals some interesting properties of the protocol verification task.

4 Time Indexing and Turchin Relation

Consider a trace $\{\Phi_i\}_{i=1}^{n}$ of a prefix rewriting grammar $\mathbf{G} = \langle \Sigma, \mathbf{R}, \Gamma_0 \rangle$. Let us write it down letter by letter, from the rightmost letter of a word to its leftmost letter. Let us mark every letter by a natural number denoting the moment when the letter appeared first (starting from the moment 0). We call this notation *time-indexing* and a trace generated by $\mathbf{G}$ with the time-indexing notation is called *a computation.*

This procedure can be described more formally as follows. The i-th letter of Φ_1 is marked by $|\Gamma_0| - i$, where $|\Gamma_0|$ is the length of Γ_0; if the largest number that is used as a time index in the track segment $\{\Phi_i\}_{i=1}^{k}$ $(k < n)$ is M and Φ_{k+1} is generated from Φ_k by the rule $R : R_l \to R_r$ then the i-th letter of Φ_{k+1} $(i \leq |R_r|$ where $|R_r|$ is the length of $R_r)$ is marked by the time index $M + |R_r| - i + 1$. Time indexes of all other letters in Φ_{k+1} remain the same as in Φ_k, since these letters are unchanged by R.

The length of a word Δ is denoted as $|\Delta|$. $\Delta[k]$ denotes the k-th letter of Δ; $\Phi \approx \Psi$ iff the words Φ and Ψ coincide up to time indices. The time-indexing notation allows take into account not only structure of a word but also the history. The time indices are useful to describe Turchin's relation on words in a trace — this relation is more powerful than Higman relation (it permits later termination of branches).

Example 9. *Consider the protocol $\mathbf{G_{RR}}$ from the example 7. The first word Γ_0 is aBa and should be annotated by the time indices as $a_{(2)}B_{(1)}a_{(0)}$ (we write the indices in the subscripts, enclosed in brackets). When it is transformed by the rule R_4, the generated word with the time indices is $b_{(3)}$. When it is transformed then by R_1, the generated word is time-indexed as $a_{(4)}b_{(3)}$.*

Definition 5. *Two words Γ and Δ form a Turchin pair iff $\Gamma = \Phi\Theta_0$, $\Delta = \Phi'\Psi\Theta_0$ and $\Phi' \approx \Phi$. This fact is denoted as $\Gamma \preceq \Delta$.*

Now the first word must not only be a subsequence of the second but also this subsequence must contain only one gap and have the special properties of the time indices. For example if the word $\Gamma = a_{(i)}$ is transformed to $\Delta = b_{(i+2)}a_{(i+1)}$ (after an application of the rule $a \to ba$), then the pair (Γ, Δ) does not satisfy Turchin relation, but it satisfies Higman one.

In 1987 V.F. Turchin proved that the Turchin pairs necessarily appear in every infinite trace generated by an arbitrary finite type 2 prefix rewriting grammar [4]. Thus the Turchin relation can be used (and is successfully used [6]) to terminate unfolding of computations in a computational tree. Our next task is to investigate for what types of prefix grammars the relation $\preceq$ can help to find at least one trace that ends with Λ and how to refine the relation to make it applicable to solve this task for any 0-type prefix grammar.

[4] A formal representation of the theorem in the terms of prefix grammars and a short proof of it can be found in [7].

5 Verification of Ping-Pong Protocols as Prefix Grammars

Now we introduce the notion of annotated prefix rewriting grammars, which was described in [7] as a way of removing all occasional Turchin pairs in traces generated by the 2-type prefix rewriting grammars.

Example 10. *Consider the following modification of the grammar* $\mathbf{G_{PA0}}$*.*

$\mathbf{G_{PAC}}$*:*

$R_1 : \Lambda \to ba$	$R_4 : b \to \Lambda$	$R_7 : \Lambda \to bac$
$R_2 : \Lambda \to a^{-1}a^{-1}$	$R_5 : a \to a^{-1}$	
$R_3 : \Lambda \to b^{-1}$	$R_6 : aa \to \Lambda$	

The two rules R_1 and R_7 are different in essence: an application of the second leads to impossibility of Λ in the further trace. But if two words Φ and its descendant Ψ form a Turchin pair, and Ψ is generated directly by R_7 from Ψ' then the word generated from Ψ' by R_1 also forms a Turchin pair with Φ. Thus $\preceq$ does not separate applications of these two rules because of the same prefix ba.

To avoid these useless Turchin pairs we can mark the letters of the right-hand side of a rule by the number of the rule in the whole list of rules and then forbid unification of the letters with different rule numbers. Such marked grammars are called annotated.

Definition 6. *A prefix rewriting grammar* $\mathbf{G}$ *is annotated if every two rules either have the same right-hand side or have no letters shared by their right-hand sides.*

Example 11. *Let us annotate the grammar* $\mathbf{G_{RR}}$ *(considering Γ_0 as R_0).*

$\mathbf{G_{ARR}}$*:*

$R_1 : \Lambda \to a^{(2)}$	$R_4 : aBa \to b^{(1)}$	$R_7 : B \to \Lambda$
$R_2 : \Lambda \to c^{(1)}$	$R_5 : aCa \to c^{(2)}$	$R_8 : C \to \Lambda$
$R_3 : \Lambda \to C^{(1)}$	$R_6 : c \to \Lambda$	$R_0 : \Lambda \to a^{(1)}B^{(0)}a^{(0)}$

The attack now looks as

$$a^{(1)}B^{(0)}a^{(0)} \to C^{(1)}a^{(1)}B^{(0)}a^{(0)} \to a^{(2)}C^{(1)}a^{(1)}B^{(0)}a^{(0)} \to c^{(2)}B^{(0)}a^{(0)} \to$$

$$\to B^{(0)}a^{(0)} \to a^{(0)} \to C^{(1)}a^{(0)} \to a^{(2)}C^{(1)}a^{(0)} \to c^{(2)} \to \Lambda.$$

We drop time indices for the sake of brevity. No pairs in this trace are comparable by $\preceq$ so the corresponding path is to be found during the unfolding.

The transition to annotated grammars leads to a refinement of $\preceq$ that allows to solve the empty word problem for the type 2 prefix rewriting grammars.

Proposition 1. *Let us consider a relation $\preceq^T$ such that $\Gamma \preceq^T \Delta$ iff $\Gamma = \Phi\Theta_0$, $\Delta = \Phi'\Psi\Theta_0$, and there exists such rule $R : R_l \to R_r$ with a non-empty right-hand side R_r that $\Phi \approx R_r$ and $\Phi' \approx R_r$. Every infinite computation generated by a 0-type prefix grammar contains an infinite subsequence $\Gamma_1, \ldots, \Gamma_n, \ldots$ such that for all n and i $\Gamma_n \preceq^T \Gamma_{n+i}$) and Γ_{n+i} is a descendant of Γ_n.*

This proposition can be reformulated in the following way. Let us say that an application of a rule $R : R_l \to R_r$ to $R_l\Theta_0$ is cancelled in a trace iff $\Theta_0[1]$ is modified or erased in the trace. Then every infinite computation by a 0-type prefix grammar contains infinite subsequence Γ_1, …, Γ_n, …, such that for all n and i Γ_n and Γ_{n+i} are generated by the same rule $R : R_l \to R_r$ with the non-empty right-hand side R_r and the application of the R in Γ_n is not cancelled in Γ_{n+i}.

Note that the usage of $\preceq^T$ with the non-annotated grammar $\mathbf{G_{RR}}$ allows to find the trace ending by Λ as well as the usage of $\preceq$ with the annotated grammar $\mathbf{G_{ARR}}$. Moreover, the transition to the annotated grammars (or to $\preceq^T$ instead of $\preceq$) solves the empty word problem for traces generated by 2-type prefix rewriting grammars.

Proposition 2. *Let* $\mathbf{G}$ *be an annotated the* 2*-type prefix rewriting grammar. If there is a computation by* $\mathbf{G}$ *ending with* Λ *then there is a computation by* $\mathbf{G}$ *ending with* Λ *and containing no Turchin pairs.*

Proof. The proof follows from the fact that every first Turchin pair in a computation by $\mathbf{G}$ satisfies also $\preceq^T$ condition. Consider the shortest computation to Λ that is generated by an annotated alphabetic prefix rewriting grammar. Suppose it contains a Turchin pair $\Gamma = R_r\Theta_0$, $\Delta = R'_r\Psi\Theta_0$. If Δ collapses to Λ then the trace from Δ to Λ must contain words $\Psi\Theta_0$ and Θ_0. But this means that there exists a way to collapse R_r to Λ and Θ_0 to Λ separately, and there is a shorter computation to Λ from Γ without reaching Δ.

□

But the strong relation $\preceq^T$ does not still solve the whole problem of finding a finite trace when verifying protocols as prefix rewriting grammars.

Example 12. *Let us try the time-indexing notation on the trace representing the attack on the protocol* P_a*.*

$$\begin{tikzcd}
\Gamma_0 : b_{(1)}a_{(0)} \arrow[d, "R_4"'] & \Gamma_2 : b_{(3)}a_{(2)}a_{(0)} \arrow[d, "R_4"'] & \Gamma_4 : \Lambda \\
\Gamma_1 : a_{(0)} \arrow[ur, "R_1"] & \Gamma_3 : a_{(2)}a_{(0)} \arrow[ur, "R_6"] &
\end{tikzcd}$$

The new a *in* Γ_3 *is marked by the index* (2) *because* (1) *is used with the first* b *which collapses in* Γ_1*.*

Even using the most powerful relation $\preceq^T$ in the example we cannot avoid the embedding $b_{(1)}a_{(0)} \preceq^T b_{(3)}a_{(2)}a_{(0)}$ which leads to the loss of attack. The bottleneck is that $\mathbf{G_{PA0}}$ is the 0-type prefix rewriting grammar. And the annotation of a grammar helps to express (and eliminate) dependencies only between the right-hand sides of rules. If a grammar is the 2-type then all the rules with the empty right-hand sides look as $x \to \Lambda$ and thus every letter can be erased independently. But if we have any dependence between the erasings, we cannot express or eliminate it by the simple annotating. Exactly this happens in $\mathbf{G_{PA0}}$: the rule $aa \to \Lambda$ assumes that we erase the first and the second a only together, but both of them can be generated only by the same rule $\Lambda \to ba$, which produces the essential Turchin pair.

Definition 7. *Let* x *be a letter that appears in the right-hand side of a rule of a given grammar. Its erasing counter is the number of the occurrences of* x *in the different left-hand sides.*

Thus, for a in $\mathbf{G_{PA0}}$ the erasing counter is 3, and for b is 1.

Example 13. *Consider the grammar* $\mathbf{G_{ARR}}$ *from Example 11.* $c^{(i)}$ *is to be erased by the single rule* R_6 *and has the erasing counter* 1*,* $a^{(i)}$ *can be erased either by* R_4 *or* R_5 *and has the erasing counter* 4*, since in the both rules there are two occurrences of* a *in the left-hand sides.*

Every letter in a 2*-type prefix grammar has the erasing counter either* 0 *(if the letter cannot be erased, as* c *in* $\mathbf{G_{PAC}}$ *from Example 10) or* 1*.*

Definition 8. *Let us say that* $\Gamma \preceq^! \Delta$ *(or forms with* Δ *a Turchin pair with erasing distinction) iff* $\Gamma \preceq^T \Delta$ *and for some* $x \in \Sigma$ *the number of the occurrences of* x *in* Δ *is greater than the erasing counter of* x*.*

Proposition 3. *Let* $\mathbf{G}$ *be a finite annotated prefix rewriting grammar. Every infinite trace generated by* $\mathbf{G}$ *either contains some* Γ *and* Δ*, such that* $\Gamma \approx \Delta$*, or contains* Γ *and* Δ*, such that* $\Gamma \preceq^! \Delta$*.*

Proof. Since there is an infinite sequence $\Gamma_1, \ldots, \Gamma_n, \ldots$, such that for all i, n $\Gamma_n \preceq^T \Gamma_{i+n}$, there exists a rule $R : R_l \to R_r$ with $R_r \neq \Lambda$, that is applied infinite many times to generate all of the Γ_i. Then the two situations can appear. Either there is an infinite number of Γ_i such that they all coincide up to the time indices, or there is a strictly growing (by the length) infinite subsequence $\{\Gamma_{n_1}\}$. Since the alphabet is finite, there exists such element Γ_{j_1} from this subsequence. that some letter from Σ repeats itself in Γ_{j_1} more times than its erasing counter is. The pair Γ_1 and Γ_{j_1} satisfies the $\preceq^!$ relation.

□

Proposition 4. *Let* $\mathbf{G}$ *be an arbitrary finite annotated prefix rewriting grammar of the type* 0*. If there is a computation by* $\mathbf{G}$ *that ends with* Λ *then there is a computation by* $\mathbf{G}$ *that ends with* Λ *and contains no Turchin pairs with erasing distinction.*

Proof. Let the shortest computation of Λ contain a Turchin pair with erasing distinction, so there are such Γ and Δ that $\Gamma \preceq^! \Delta$. Let a be a letter that is to be erased twice by the same rule. Consider the immediate moments before the erasings. They must look like $\hat{R_l} a \Psi a \Theta_0$ and $\hat{R_l}' a \Theta_0$, where $\hat{R_l}$ denotes a prefix of the left-hand side of some rule. Now consider the moments in which a were generated. They must look like $\hat{R'_r} a \Theta_0$ and $\hat{R'_r} a \Psi a \Theta_0$ respectively and form the pair in respect of $\preceq^!$.

All the considered moments together form the following sequence.

$\ldots$
$\hat{R'_r} a \Theta_0$
$\ldots$
$\hat{R'_r}' a \Psi a \Theta_0$
$\ldots$
$\hat{R_l} a \Psi a \Theta_0$
$\ldots$
$\hat{R_l}' a \Theta_0$
$\ldots$
Λ

Now we can apply to $\hat{R'_r} a \Theta_0$ the same transformations as in the segment from $\hat{R'_r}' a \Psi a \Theta_0$ to $\hat{R_l} a \Psi \Theta_0$ and get a shorter computation to Λ.

□

Proposition 4 together with the proposition 3 gives a sound condition of when to terminate a computation if we want to find the shortest finite computation path. If some two words that are comparable over $\preceq^!$ or over $\approx$ appear, then we can stop unfolding the computation that contains them. All infinite computations contain such pairs, so we cannot diverge when unfolding until them.

The proposition 4 implies an obvious observation that if there a rule with a letter in the right-hand side that have the erasing counter 0 is applied in a computation then the computation contains no Λ. Moreover, the proposition 2 is a straight consequence of the lemma 4 if none letters in rules with non-empty right-hand sides have erasing counters greater than 1.

6 Modelling Long Attacks via Refined Turchin Relation

Now when we know the test of branch termination for finding attacks, the question of maximal attack length arises. The work [7] describes how to build maximal trace with no Turchin pairs in it. But this trace is not necessarily a minimal trace ending with Λ.

Example 14. *Consider the annotated grammar from [7]. Remind that R_0 coincides with the initial word Γ_0. x denotes an arbitrary letter from $\Sigma = \{a, b, c, d, e, f, g\}$.*

$$\begin{array}{ll}
\mathbf{G'_F}\textit{:} & \\
R_0 : x \to ab & R_2 : x \to ef \\
R_1 : x \to cd & R_3 : x \to g \\
R_4 : x \to \Lambda &
\end{array}$$

The longest trace of this grammar without the Turchin pairs has the length 22. However, the shortest attack has the length 3: it is enough to apply rule R_4 twice to the initial $a_{(1)}b_{(0)}$ to receive Λ.

Now our task is not only to build the longest possible trace with no Turchin pairs but to build a grammar with a long shortest possible trace ending by Λ (in particular, this means it contains no pairs in respect of $\preceq^!$ but very likely contains some pairs in respect of $\preceq^T$). We use the same idea of the "ladder" construction as in [7] to show that it is possible to construct a grammar with a minimal trace ending by Λ such that it repeats the "ladder" constructions as well as contains pairs over $\preceq^T$.

Example 15. *In the following grammar $\mathbf{G_{EXP}}$ the shortest trace ending by Λ has the length 80, see Figure 1.*

$$\begin{array}{lll}
\mathbf{G_{EXP}}\textit{:} & & \\
R_1 : \Lambda \to aA & R_4 : Ba \to bB & R_7 : BB \to cC \\
R_2 : \Lambda \to bB & R_5 : AA \to bB & R_8 : c \to \Lambda \\
R_3 : \Lambda \to cC & R_6 : Cb \to cC & R_9 : CC \to \Lambda
\end{array}$$

7 Conclusion

We described a way to verify some class of ping-pong protocols via unfolding techniques and prefix rewriting grammars. Now we considered not only the 2-type rewriting grammars but grammars of the type 0 of Caucal hierarchy. We showed that the condition of path termination with respect of $\preceq^!$ cannot be replaced by $\preceq^T$ or weaker conditions (such as Higman relation) and how a grammar with a small number of rules can generate a long minimal finite computation. Thus the question of how to verify systems described by the 0-type prefix rewriting grammars via computational tree unfolding is answered in general.

The long trace ending by Λ generated by $\mathbf{G_{EXP}}$ with the initial word aA.

1	aA	$cCBBA$	$cCbBAA$	bB
2	$bBaA$	$CBBA$	$CbBAA$	$cCbB$
3	$cCbBaA$	$cCCBBA$	$cCBAA$	CbB
4	$CbBaA$	$CCBBA$	$CBAA$	cCB
5	$cCBaA$	BBA	$cCCBAA$	CB
6	$CBaA$	cCA	$CCBAA$	$cCCB$
7	$cCCBaA$	CA	BAA	CCB
8	$CCBaA$	$cCCA$	$bBBAA$	B
9	BaA	CCA	$cCbBBAA$	bBB
10	bBA	A	$CbBBAA$	$cCbBB$
11	$cCbBA$	aAA	$cCBBAA$	$CbBB$
12	$CbBA$	$bBaAA$	$CBBAA$	$cCBB$
13	$cCBA$	$cCbBaAA$	$cCCBBAA$	CBB
14	CBA	$CbBaAA$	$CCBBAA$	$cCCBB$
15	$cCCBA$	$cCBaAA$	$BBAA$	$CCBB$
16	$CCBA$	$CBaAA$	$cCAA$	BB
17	BA	$cCCBaAA$	CAA	cC
18	$bBBA$	$CCBaAA$	$cCCAA$	C
19	$cCbBBA$	$BaAA$	$CCAA$	cCC
20	$CbBBA$	$bBAA$	AA	CC

Figure 1: The Long Attack on $\mathbf{G_{EXP}}$

This shows that the field of applications of the Turchin relation is much wider than function stack embeddings, as in the original work. Annotation and erasing distinction can be a reliable method of avoiding too early termination of unfolding, but for many special classes of systems it would be more efficient to use not $\preceq^{!}$ itself but its generalizations or restrictions that are more relevant to the class.

References

[1] M. Abadi and A.D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998.

[2] A.C. Yao D. Dolev. On the security of public key protocols. *Transactions on Information Theory*, 29:198–208, 1983.

[3] S. Even D. Dolev and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55:57–68, 1982.

[4] G. Delzanno, J. Esparza, and J. Srba. *Monotonic Set-Extended Prefix Rewriting and Verification of Recursive Ping-Pong Protocols*, volume 4218 of *Lecture Notes in Computer Science*, pages 415–429. IEEE Computer Society Press, 2006.

[5] P. Jancar and J. Srba. *Undecidability Results for Bisimilarity on Prefix Rewrite Systems*, volume 3921 of *Lecture Notes in Computer Science*, pages 277–291. IEEE Computer Society Press, 2006.

[6] A. P. Nemytykh. *The Supercompiler Scp4: General Structure.* URSS, Moscow, 2004.

[7] A. Nepeivoda. A refinement of higman embedding for loop approximation. [unpublished], 2013. `http://refal.botik.ru/preprints/Antonina_Nepeivoda-On_Turchin_Theorem-06042013v1.pdf`.

[8] M. H. Sørensen and R. Gluck. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

[9] M.H. Sørensen, R. Gluck, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:465–479, 1993.

[10] V.F. Turchin. The algorithm of generalization in the supercompiler. *Partial Evaluation and Mixed Computation*, pages 341–353, 1988.