User's Guide for the LySatool version 2.01

Mikael Buchholtz

April 18, 2005

1. Introduction

The LySatool [4] implements a security analysis for the process calculus LySa. This calculus comprises language primitives for modelling of network communication that achieves security by applying cryptographic techniques. The analysis is an over-approximating program analysis that can guarantee *confidentiality* and *authentication* properties for LySa processes — even when these are under attack from arbitrary network attackers.

This user's guide explains how to operate the LySatool. However, it does *not* explain how the analysis works nor does it give the necessary insight to interpret the analysis result. The user is referred to other published material on these matters. For example, LySa and the analysis are described in [1]. The main novelty of version 2 of the LySatool is an addition of an analysable meta-level, which is described in [3]. Further details on the correctness of the various features implemented in LySatool version 2 may be found in [2].

2. Operating the LySatool

The LySatool is implemented in Standard ML of New Jersey [5] (SML/NJ) and uses the Succinct Solver [6] as a solving back end. The LySatool takes as input a LySa process and returns an analysis result. The LySatool can be invoked by running an SML/NJ script as detailed below. Alternatively, the LySatool can be invoked through calls to an SML/NJ API, which contains a rich set of function for interacting with the tool. The API is surveyed in Appendix A.

2.1. A Sample Session

The LySatool can be invoked through the script run.sml, which can be loaded into an SML/NJ promt by

```
> use("run.sml");
```

The input to the LySatool is a file containing a LySa process. The script contains a function `run` to invoke the tool on such a file. For example, the LySatool may be invoked on the file examples/publickey.lysa by calling

```
> run("examples/publickey.lysa");
```

The result of running the LySatool is an HTML file examples/publickey.html, which contains the input LySa process along with the analysis result. Both are presented in HTML format. The resulting file has the same name as the input file except that its suffix is replaced by .html.

The LySatool can be invoked with various parameters. This can by done by calling the function `runp`, which takes parameters as its second argument. For example,

```
> runp("examples/publickey.lysa", Analysis2.version1parameters);
```

runs the LySatool with parameters that correspond to the behaviour of version 1 of the LySatool. The syntax for LySa processes is detailed in Section 3 while the parameters are discussed in Section 4

3. ASCII Grammar of LySa

The LySatool takes as input a LySa process, which may contain destination and origin authentication annotations. The LySa process is given in ASCII format following the grammar in Figure 1. Here the set *identifier* contains strings that begin with a letter followed by zero, one, or more letters or digits. The set *number* contains strings with decimal representation of non-negative numbers. The following list of strings are considered to be *keywords* that cannot be used for other purposed than described by the grammar:

> `as`, `at`, `CPDY`, `dest`, `decrypt`, `in`, `let`, `orig`, `NATURAL1`, `NATURAL2`, `NATURAL3`, `NATURAL01`, `NATURAL02`, `NATURAL03`, `new`, `subset`, `ZERO`

Any string between an opening `/*` until the first closing `*/` are comments and disregarded when parsing.

In the grammar a non-terminal post-fixed with $^+$ denotes non-empty comma-separated list of that non-terminal while a non-terminal post-fixed with * denotes a (possibly empty) comma-separated list of that non-terminal. An ε in the body of a rule denotes the empty string.

The prefix operators on processes (input, output, decryption, and restriction) binds tighter than the `!` at replication, which again binds tighter than parallel composition. Parallel composition is left associative. Finally, indexed parallel has lowest precedence meaning that the scope of indexed parallel reaches as far as possible! Alternative precedence may be forced by adding parentheses around processes and terms.

proc	:: =	`(` *proc* `)` | `<` *term** `>` `.` *proc* | `(` *term** `;` *var** `)` `.` *proc* | `decrypt` *term* `as` `{` *term** `;` *var** `}` `:` *term orig* `in` *proc* | `decrypt` *term* `as` `{|` *term** `;` *var** `|}` `:` *term orig* `in` *proc* | `(` `new` *name* `)` *proc* | `(` `new` `+-` *name* `)` *proc* | `!` *proc* | *proc* `|` *proc* | `0` | `let` *identifier* `subset` *iset* `in` *proc* | `|_{` *assign* `}` *proc* | `(` `new_{` *assign*$^+$ `}` *name* `)` *proc* | `(` `new_{` *assign*$^+$ `}` `+-` *name* `)` *proc*
term	::=	`(` *term* `)` | `{` *term** `}` `:` *term dest* | `{|` *term** `|}` `:` *term dest* | *name* | *namep* | *namem* | *var*
name	::=	*identifier subscript*
namep	::=	*identifier* `+` *subscript*
namem	::=	*identifier* `-` *subscript*
var	::=	*identifier subscript*
subscript	::=	`_{` *index** `}` | ε
index	::=	*identifier* | *number*
iset	:: =	`{` index* `}` | *iset* `union` *iset* | `NATURAL1` | `NATURAL2` | `NATURAL3` | `NATURAL01` | `NATURAL02` | `NATURAL03` | `ZERO`
assign	::=	*index* `in` *number*
dest	::=	`[` `at` *cryptopoint* `dest` `{` *cryptopoint** `}` `]` | `[` `at` *cryptopoint* `]` | ε
orig	::=	`[` `at` *cryptopoint* `orig` `{` *cryptopoint** `}` `]` | `[` `at` *cryptopoint* `]` | ε
cryptopoint	::=	*identifier subscript* | `CPDY`

Figure 1: ASCII grammar for LySa.

Names and variables are parsed according the same syntax and conflicts are resolved by ensuring that any occurrence of a variable that is in scope (of an input or a decryption) will indeed be interpreted as a variable. At all other places elements will be interpreted as names. Thus, processes will never contain free variables though they may contain free names.

Annotations of origin and destination information are added in square brackets. The annotations give both a crypto-point denoting the place of decryption/encryption as well as a set of crypto-points for expected origin/destination. The latter information may be left out and this will be interpreted as the semantic equivalent of having no requirement of origin/destination. If an annotation is altogether empty the decryption/encryption will further more be interpreted as if it takes place at an unspecified crypto-point.

4. LySatool Parameters

The LySatool takes the following parameters in an SML record (where default values are given in parenthesis):

`withAttacker: bool` (`true`) input process is analysed together with all arbitrary attackers when `withAttacker` is set to `true`. With to `false`, the analysis is run without an attacker

`attackerIndex: string` (`""`) It is sometimes desirable to model that legitimate principals communicate with the attacker (i.e. with illegitimate principals). If the legitimate principals are described by an indexed parallel composition it may be convenient to let a particular index (e.g. index 0) denote interaction with the attacker. However, with this kind of modelling, interaction with the attacker may cause auxiliary error messages in ψ because the attacker uses `CPDY` as its crypto-point but `CPDY` is usually not an intended origin or destination specified by the annotations.

By assigning the parameter `attackerIndex` the value `"0"` each set of crypto-points in an annotation of the 0^{th} unfolding of an index parallel composition will have `CPDY` added. Thus, interaction with the attacker, which occurs in 0^{th} unfolding, will no longer cause auxiliary error messages.

`mergeExpressionLabels: bool` (`true`) Labels are automatically attached to all LySa expressions and these labels are used as non-terminals in the tree grammars in the analysis result. When `mergeExpressionLabels` is set set to `true`, identical expressions may be assigned the same label. This, in turn, may reduce the size of the tree grammars that are computed, thereby, reducing the time it takes to compute the analysis result. When set to `false` labels are unique.

`mergeInputVariables: bool` (`true`) When the attacker is present, all the values known to the attacker may become bound to any variables in an input. When the parameter `mergeInputVariables` is set to `true` all these *input variables* are merged into a single variable, $x_\bullet$, in the analysis result. This may reduce the size of the analysis result, thereby, reducing the time it takes to compute the result. It only makes sense to set this parameter to `true` when the attacker is present!

References

[1] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 2004. To appear. Preliminary version available at http://www.imm.dtu.dk/pubdb/views/edoc_download.php/3199/pdf/imm3199.pdf.

[2] M. Buchholtz. Automated analysis of security in networking systems. Ph. D. thesis proposal. Available from http://www.imm.dtu.dk/~mib/thesis/, December 2004.

[3] M. Buchholtz. Automated analysis of infinite scenarios. In *Proceedings of Trustworthy Global Computing (TGC 2005)*, Lecture Notes in Computer Science. Springer Verlag, 2005. To appear.

[4] LySatool. http://www.imm.dtu.dk/cs_LySa/lysatool, 2005. Webpage hosted by Informatics and Mathematical Modelling, Technical University of Denmark.

[5] Standard ML of New Jersey. http://www.smlnj.org/, 2004. Webpage hosted by the SML/NJ Fellowship.

[6] Succinct Solver. http://www.imm.dtu.dk/cs_SuccinctSolver/, 2004. Webpage hosted by Informatics and Mathematical Modelling, Technical University of Denmark.

A. API

The LySatool distribution contains the following groups of files:

run.sml	script to run the LySatool directly on a file and produce HTML output.
mllysa.sml	data structure `MLLysa`, which represents LySa processes, and auxiliary functions on processes.
analysis2.sml	data structure `Analysis2` for the constraint generation function of the meta-level analysis (see Figure 2).
io/	directory for input/output functionally from/to ASCII (see Figure 3), to HTML (see Figure 4), to LaTeX (see Figure 5).
io/lysa.sty	LaTeX package for typesetting LySa processes.
io/lysa-mode.el	simple Emacs mode for syntax high-lighting of LySa processes.
examples/	directory for examples.
sources.cm	source file for SML/NJ Compilation Manager.
COPYRIGHT	copyright for the LySatool.

A.1. `Analysis2`

The constraint generation function is embedded in the structure `Analysis2` shown in Figure 2. The constraint generation is performed by the function `generate`, which takes a LySa process and parameters as input and produces a ALFP constraint as output. The parameters are detailed in Section 4 and may be `default`, `version1parameters`, or user specified parameters. The function `analyse` is provided for backwards portability with version 1 of the LySatool. The string `version` specifies the current version of the LySatool.

```
structure Analysis2 :
  sig
    val generate : {attackerIndex:string, mergeExpressionLabels:bool,
                    mergeInputVariables:bool, withAttacker:bool}
                   -> MLLysa.Proc -> string
    val default : {attackerIndex:string, mergeExpressionLabels:bool,
                   mergeInputVariables:bool, withAttacker:bool}
    val version1parameters : {attackerIndex:string, mergeExpressionLabels:bool,
                              mergeInputVariables:bool, withAttacker:bool}
    val analyse : MLLysa.Proc -> string

    val version : string
  end
```

Figure 2: Structure for constraint generation in the file analysis2.sml.

```
structure LysaASCIIIO :
  sig
    exception parseError of string
    val parseFile : string -> MLLysa.Proc
    val parseStream : TextIO.instream -> MLLysa.Proc
    val parseString : string -> MLLysa.Proc
    val toFile : string -> MLLysa.Proc -> unit
    val toString : MLLysa.Proc -> string
  end
```

Figure 3: Structure for ASCII input and output in the file io/lysaasciiio.sml.

A.2. LysaASCIIIO

The structure `LysaASCIIIO` shown in Figure 3 can be used to parse LySa processes from a file, stream, or a string according to the grammar given in Section 3. The structure also contains functions `toString` and `toFile` to print a LySa process in ASCII format into a string and a file, respectively.

A.3. LysaHTMLIO

The structure `LysaHTMLIO` shown in Figure 4 contains a function `toHTML`, which takes a file name (without suffix), a LySa process, the analysis parameters, and the analysis result. It produces a browsable HTML document (suffixed .html) containing the LySa process and the analysis result.

A.4. LysaLatexIO

The structure `LysaLatexIO` shown in Figure 5 contains functions to print LySa processes into LaTeX format. This LaTeX output uses macros from the LaTeX package in the file

```
structure LysaHTMLIO :
  sig
    val toHTML : string
                 -> MLLysa.Proc
                    -> {attackerIndex:string, mergeExpressionLabels:bool,
                        mergeInputVariables:bool, withAttacker:bool}
                       -> 'a * (string * string list list) list -> unit
  end
```

Figure 4: Structure for HTML output of processes and analysis results in the file io/lysahtmlio.sml.

```
structure LysaLatexIO :
  sig
    val toFile : string -> MLLysa.Proc -> unit
    val toString : MLLysa.Proc -> string
  end
```

Figure 5: Structure for LaTeX output of processes in the file io/lysalatexio.sml.

io/lysa.sty.