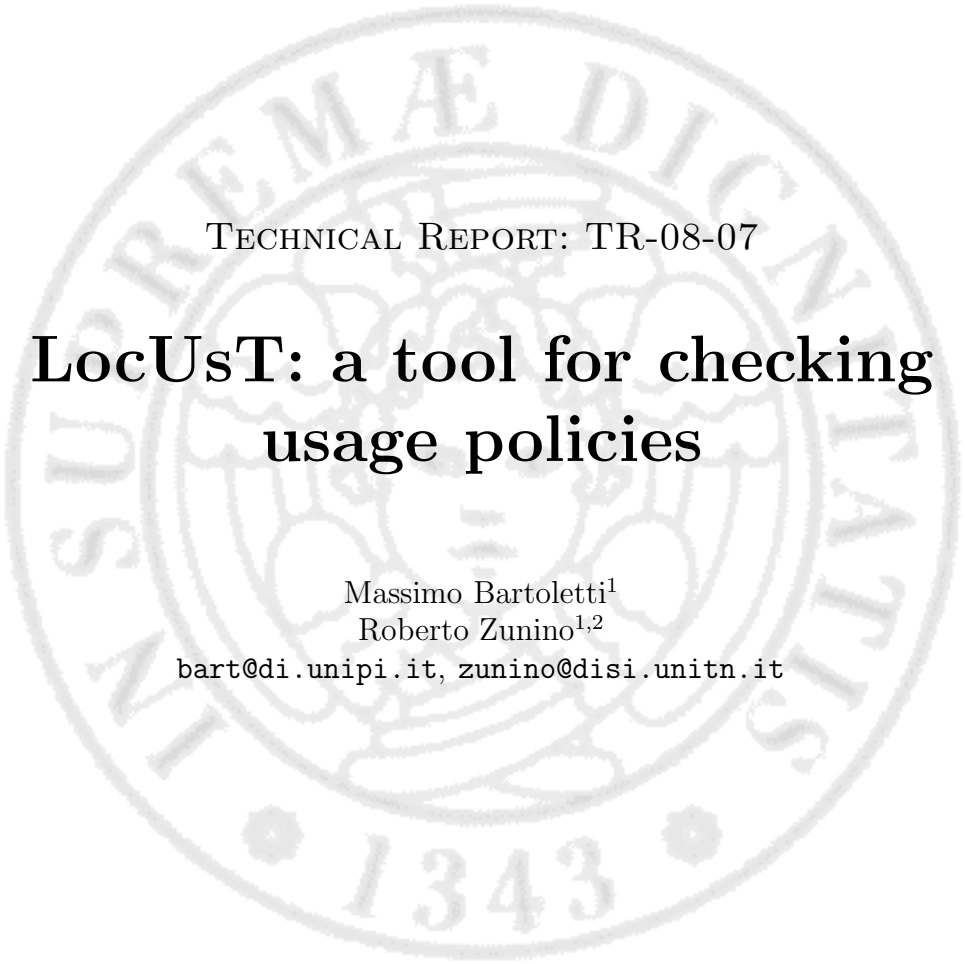
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-08-07

LocUsT: a tool for checking usage policies

Massimo Bartoletti[1]
Roberto Zunino[1,2]
bart@di.unipi.it, zunino@disi.unitn.it

April 4, 2008

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

LocUsT: a tool for checking usage policies

Massimo Bartoletti[1] and Roberto Zunino[1,2]
bart@di.unipi.it, zunino@disi.unitn.it

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy

Abstract. We introduce LocUsT, a tool to statically check whether a given resource usage complies with a local policy.
LocUsT takes as input an abstraction of the behaviour of a program, called a *usage*. Usages are expressed in a simple process calculus, and over-approximate all the resource accesses of the program itself. As additional input, LocUsT takes a policy that defines the allowed resource access patterns, represented through a finite state automaton parametrized over resources. Finally, LocUsT decides whether some trace of the given usage violates some instantiation of the policy.

1 Introduction

Local policies were first introduced in [1]. There, a block of code can be sandboxed by a local policy framing $\varphi[B]$ so to require that the policy φ must hold while B is executed: after B terminates, there is no such requirement – hence *local* policies. Notably, policies can inspect the whole execution *history* generated so far, e.g. they can forbid the program to send an e-mail if private data was read in the past. In [1], some key concepts and techniques appeared, though, in their original formulation usages were only able to express the performed action (e.g. read or write) without specifying the target resource of the action (e.g. file_1 or file_2). Similarly, policies defined regular set of action strings, neglecting resources.

Dealing with resources was the main topic of [4]. There, actions were augmented with resources, and the whole model was changed to reflect that. First, policies were parametrized over resources: φ can now be written as a finite state automaton which depends on the resource x at hand. Accordingly, $\varphi[B]$ now requires that *all* the resources satisfy $\varphi(x)$ during the execution of B. Finally, to keep our model realistic, an unbounded number of resources can be dynamically generated by usages.

The model proved itself rather general and capable of modelling several interesting policies. For instance, secure orchestration of web services and *call by contract* has been the main concern of [2, 6, 3].

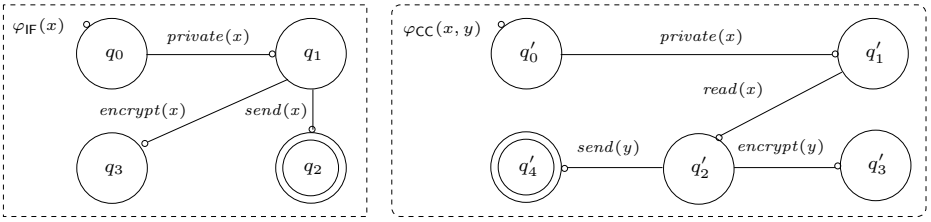
The LocUsT tool implements the verification techniques for resource usage described in the papers discussed above. Given a usage and a policy, the tool decides whether all the (possibly infinite) traces generated by the usage comply with the policy (for all the instantiation of its parameters).

This paper outlines the main features of the LocUsT tool. The underlying theory can be found in [5]. We remand the reader there for the actual definition of usages and policies, altough we expect that a reader with some background in process calculi should be able to understand the examples shown in this paper.

Noteworthy features of LocUsT include the handling of dynamic resource creation, and the support for polyadic events (e.g. send(msg, recipient)) and polyadic policies (e.g. $\varphi(x, y)$).

2 Examples

We borrow the following policy examples from [5].



The automata above show an information flow policy $\varphi_{\mathsf{IF}}(x)$ and a covert channels policy $\varphi_{\mathsf{CC}}(x, y)$. The policy $\varphi_{\mathsf{IF}}(x)$ states that, after a resource x is flagged as private, it can not be sent over the network unless encrypted – this avoids direct information flows. The policy $\varphi_{\mathsf{CC}}(x, y)$ requires that, after having read private data, no unencrypted data (at all) can be sent – this avoids indirect information flows. We refer to [5] for more discussion.

Here is the policy $\varphi_{\mathsf{IF}}(x)$ expressed with LocUsT syntax:

```
name: phi_IF
states: q0 q1 q2 q3
start: q0
final: q2
trans:
q0 -- private(x) --> q1
q1 -- encrypt(x) --> q3
q1 -- send(x)    --> q2
```

Here[1] is the policy $\varphi_{\mathsf{CC}}(x, y)$:

```
name: phi_CC
states: q0 q1 q2 q3 q4
start: q0
final: q4
```

[1] Currently, LocUsT parameters must begin with the letter `x` to be distinguished from constants, so we use `x1` and `x2` in the source

```
trans:
q0 -- private(x1) --> q1
q1 -- read(x1)    --> q2
q2 -- send(x2)    --> q4
q2 -- encrypt(x2) --> q3
```

And here are the results for sone selected usages:

`phi_IF[ nu n. private(n) . ( send(n) + encrypt(n) ) ]`	**FAIL**
`phi_IF[ nu n. nu f. private(n) . read(n) . send(f) ]`	**PASS**
`phi_CC[ nu n. nu f. private(n) . read(n) . send(f) ]`	**FAIL**
`nu n. private(n) . nu f. (mu h. phi_CC[ send(f) ] + read(n) . h)`	**FAIL**
`nu n. private(n) . nu f. (mu h. phi_CC[ send(f) . h ] + read(n) . h)`	**FAIL**
`nu n. private(n) . nu f. (mu h. phi_CC[ send(f) . h ] + read(n) . encrypt(f) . h)`	**PASS**

3 The Verification Technique

We now briefly recap the verification technique described in detail in [5], which is the one implemented in the LocUsT tool. This is not meant to be a full description, but merely a quick overview, as well as a guide to understand how the algorithms of [5] are composed.

- **Regularization.** First, the usage is *regularized*, i.e. transformed so that in no trace a policy framing $\varphi[-]$ is entered twice: for instance $\varphi[U \cdot \varphi[U']]$ becomes $\varphi[U \cdot U']$. Particular care must be exercised when handling recursive usages such as $\mu h.\varphi[h + U]$.
- **Conversion into BPA.** The usage is transformed in a process of Basic Process Algebras. Notably, here dynamic creation caused by νn is handled by instantiating n with suitable static witnesses. This step is correct, but introduces some spurious traces that might mine completeness: e.g. in some trace of the BPA associated to $(\nu n.U) \cdot (\nu m.U')$ the witness of n and m are chosen to be the same. These BPA traces actually have no usage counterpart, so we can safely ignore them. Indeed, we shall discard them later to recover completeness.
- **Framing the Policy.** The policy automaton is duplicated so that the first copy handles the transitions made by the usage when *outside* the policy framing, and the second copy handles them when *inside* the policy framing.
- **Instantiating the Policy.** The (possibly polyadic) policy is instantiated, non-deterministically assigning to each parameter a statically known resource or one of the static witnesses used above.
- **Weak Until.** Policy automata are adapted so to ignore traces where the same witness # appears to be generated twice, i.e. those having a double *new*(#) event.

– **Model Checking.** Finally, the language of the BPA is compared with the transformed policy. With a model checking algorithm we decide whether a violation is possible. Here we model-check all the instantiated policies produced before. Note that a non-terminating BPA such as $\mu h.\ U \cdot h$ can still cause policy violations, despite having no finite trace. Hence, here we must also consider the prefixes of the infinite traces as well.

The complexity of LocUsT is polynomial in the size of the usage and the size of the policy. There is an exponential factor in the number of policy parameters, only. From a pragmatic point of view, we expect the number of parameters to be very small in practice, so this should be of small concern. This exponential factor is mainly due to the policy instantiation step above, which is non-deterministic.

4 Availability

The LocUsT tool is free software, and is written in Haskell. The current version can be found at http://www.di.unipi.it/∼zunino/software/locust.

5 Acknowledgements

This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers).

References

1. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.
2. M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. In *Proc. 19th CSFW*, 2006.
3. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Secure service orchestration. In *FOSAD*, pages 24–74, 2007.
4. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proc. Fossacs*, 2007.
5. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. Technical Report TR-08-06, Dip. Informatica, Univ. Pisa, 2008.
6. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Semantics-based design for secure web services. *IEEE Transactions on Software Engineering*, 34(1), 2008.