Do you really mean what you actually enforced?
Edited automata revisited

Nataliia Bielova · Fabio Massacci

Published online: 24 June 2011
© Springer-Verlag 2011

Abstract In their works on the theoretical side of Polymer, Ligatti and his co-authors have identified a new class of enforcement mechanisms based on the notion of edit automata that can transform sequences and enforce more than simple safety properties. We show that there is a gap between the edit automata that one can possibly write (e.g., by Ligatti et al in their IJIS running example) and the edit automata that are actually constructed according the theorems from Ligatti’s IJIS paper or from Talhi et al. “Ligatti’s automata” are just a particular kind of edit automata. Thus, we re-open a question which seemed to have received a definitive answer: you have written your security enforcement mechanism (aka your edit automata); does it really enforce the security policy you wanted?

Keywords Formal models for security · Trust and reputation · Resource and access control · Validation/Analysis tools and techniques

1 Introduction

The explosion of multi-player games, P2P applications, collaborative tools on Web 2.0, and corporate clients in service-oriented architectures, has changed the usage models of PC users: users demand to install more and more interactive applications from a variety of sources. Unfortunately, the features of those applications are at odds with the current security model.

The first hurdle is certification. Certified application by trusted parties can run with full powers while untrusted ones essentially without any powers. However, certification just says that the code is trusted rather than trustworthy because the certificate has no semantics whatsoever. Will your apparently innocuous application collect your private information and upload it to a remote server [19]? Will your corporate client developed in out-sourcing dump your hard disk in a shady country? You have no way to know.

Model carrying code [21] or Security-by-Contract [3] which claim that code should come equipped with security claims to be matched against the platform policies could be a solution. However, this will only be a solution for certified code.

To deal with the untrusted code either .NET [14] or Java [10] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous functionalities, such as starting connections or accessing sensitive information. The drawback is that after assigning a permission, the user has very limited control over its usage. An application with a permission to upload a video can then send hundreds of them invisibly for the user (see the Blogs on UK Channel 4’s Video on Demand application [7]). Conditional permissions that allow and forbid use of the functionality depending on such factors as the bandwidth or some previous actions of the application itself are currently out of reach. The consequence is that either applications are sandboxed (and thus can do almost nothing) or the user decided that they are trusted and then they can do almost everything.

To overcome these drawbacks, a number of authors have proposed to enforce the compliance of the application to the user’s policies by execution monitoring. This is the idea behind security automata [1,8,11,20], safety control of
Java programs using temporal logics [12] and history-based access control [13].

In order to provide enforcement of security policies by runtime monitoring untrusted programs, we want to know what policies are enforceable and what mechanisms can actually enforce them. In a landmark paper [2], Bauer, Ligatti, and Walker seemed to provide a definitive answer by presenting a new hierarchy of enforcement mechanisms and a classification of security policies that are enforceable by these mechanisms.

Traditional security automata were essentially action observers that stopped the execution as soon as an illegal sequence of actions was on the eve of being performed. The new classification of enforcement mechanisms proposed by Ligatti included truncation, insertion, suppression, and edit automata which were considered as execution transformers rather than execution recognizers. The great novelty of these automata was their ability to transform the “bad” program executions in good ones.

These automata were then classified with respect to the properties they can enforce: precisely and effectively enforceable properties. It is stated in [2] that as precise enforcers, edit automata have the same power as truncation, suppression, and insertion automata. As for effective enforcement, it is said that edit automata can insert and suppress actions by defining suppression-rewrite and insertion-rewrite functions and thus can actually enforce more expressive properties than simple safety properties. The proof of Theorem 8 in [2] provides us with a construction of an edit automaton that can effectively enforce any (enforceable) property.

Talhi et al. [22] have further refined the notion by considering bounded version of enforceable properties.

1.1 Contribution of the paper

If everything is settled why are we writing this paper? Everything started when we tried to formally show “as an exercise” that the running example of edit automaton from [2] provably enforces the security policy described in that paper by applying the effective enforcement theorem from the very same paper. Much to our dismay, we failed.

As a result of this failure, we decided to plunge into a deeper investigation and discovered that this was not for lack of will, patience, or technique. Rather, the impossibility of reconciling the running example of a paper with the theorem on the very same paper is a consequence of a gap between the edit automata that one can possibly write (e.g., by Ligatti himself in his running example) and the edit automata that are actually constructed following Theorem 8 from [2] or Theorem 3.3 from [15] or Talhi et al. [22]. The edit automata constructed according to those theorems are just a particular kind of edit automata. We named them “Ligatti” automata.

In Fig. 1, we show the relation between different classes of automata we are investigating in this paper. All-Or-Nothing automata at every step output the whole input sequence or suppress the current action. The notion of effective enforcement is taken from [2]. Late automata are a particular kind of edit automata that always output some prefix of the input.

Figure 8 later in the paper shows the relations among different classes of edit automata, even though they are the “same” according to [2].

The contribution of this paper is therefore manifold:

- We introduce a fine-grained classification of edit automata and related security properties and relation between different notions of enforcement.
- We show the difference between the running example from [2] and the edit automata that are constructed according to the Theorem 8 in the very same paper.
- We further explain the gap by showing that the particular automata constructed according to Theorem 8 in [2] are a particular form of late automata that have an all-or-nothing behavior (Ligatti automata).
- We show that the construction from Talhi et al. [22] only applies to Ligatti automata and provides a more useful construction that is the inverse of Talhi et al. [22] construction: namely from a policy specification expressed as a Büchi automaton, we show how to construct a Ligatti’s automaton that enforces it.

The remainder of the paper is structured as follows. At first we sketch the difference between the edit automaton from the running example and Theorem 8 from [2] (§2). Then we present the basic notions of policies, enforcement, and automata in Sect. 3. We give a more fine-grained classification of edit automata introducing the notion of Late automata (§4). Sect. 5 explains relation between different notions of enforcement and types of edit automata. We provide the construction of Ligatti’s automaton that enforces a policy.