Towards Verifiable Trust Management for Software Execution
(Extended Abstract)

Michael Huth and Jim Huan-Pu Kuo

Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth,jimhkuo}@imperial.ac.uk

Abstract. In the near future, computing devices will be present in most artefacts, will considerably outnumber the number of people on this planet, and will host software the executes in a potentially hostile and only partially known environment. This suggests the need for bringing trust management into running software itself, so that executing software be guard-railed by policies that reflect risk postures deemed to be appropriate for software and its deployment context. We sketch here an implementation of a prototype that realizes, in part, such a vision.

The technical work described below relies on the concept of *Trust Evidence*. By this we mean any source of information (credentials, reputation, system state, past or present behavior, etc.) that can be used in order to assess the trustworthiness of running a unit of code. The variety of sources for Trust Evidence suggest the need for an extensible language in which such evidence can be combined. The quantitative (e.g. reputation) and qualitative (e.g. a claimed credential) nature of such evidence means that such a language has to consistently compose qualitative as well as quantitative notions of Trust Evidence.

We here present an exploratory case study (whose usability issues are discussed in [1]) where Scala [2] methods are the units of software that guard rails are meant to control. Guard rails use heterogeneous Trust Evidence sources to decide the circumstances in which methods may be invoked. The data-flow diagram of our case study, in Figure 1, has a three-layered guard rail architecture.

Annotation blocks `@Expects`, `@Policy`, and `@Switch` precede each method declaration and, roughly, correspond to a *context-sensitive access request*, a *policy decision point* (where the contextualized request is evaluated), and a *policy enforcement point* (where the evaluated decision is realized) – as familiar from access-control architectures. Atomic expectations (expressed in `@Expects`) may be predicates associated with a certain trust score. Intuitively, truth of the predicate secures this trust score in isolation. For example, an atomic expectation may assign trust score 0.2 if the method is not called by a specific caller method. Atomic trust scores would then be composed within sub-blocks based on a specified composition operator (e.g. a pessimistic min operator). Sub-block scores may be accumulative, pessimistic, etc. and are themselves combined into a local trust score that is referred to in the second level (`@Policy`), which specifies

M. Huth et al. (Eds.): TRUST 2013, LNCS 7904, pp. 275–276, 2013.

a *rule-based policy* for whether or not to execute the method body. The third level (`@Switch`) then implements the *enforcement of the guard rails* specified in the expectation and policy levels. We implemented this as a switch statement that ranges over possible policy decisions and that specifies, for each possible decision, whether and if so how the "payload" method body should execute.

The proof of concept framework was implemented in the Scala programming language [2]. One reason for choosing a JVM based language such as Scala is that there exist frameworks to extend the language and inject behaviours/aspects at various points of the source code, allowing us to deliver more refined, future prototype implementations. The two frameworks we use in our first implementation are: ANTLR [3] and AspectJ [4].

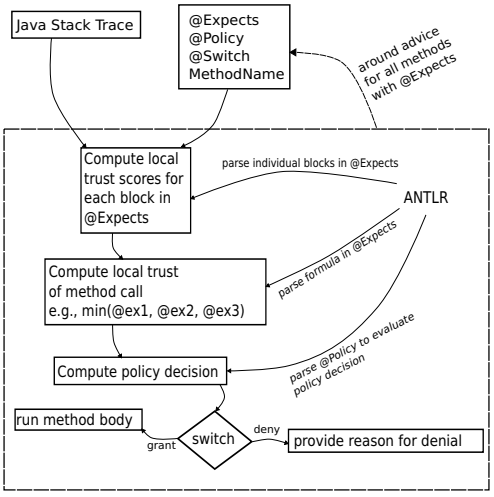

Fig. 1. Dataflow of our trust-management implementation for method guard railing

Acknowledgment. Intel® Corporation kindly funded a sub-project within its *Trust Evidence* project. Work reported here is an outcome of said sub-project.

References

1. Huth, M., Kuo, J.H.-P., Sasse, A., Kirlappos, I.: Towards usable generation and enforcement of trust evidence from programmers' intent. In: Proc. of 15th Int'l Conf. on Human-Computer Interaction. LNCS. Springer (to appear, 2013)
2. Odersky, M.: The Scala Language Specification Version 2.9. Programming Methods Laboratory, EPFL, Switzerland (May 24, 2011) (draft)
3. Parr, T.: The Definitive ANTLR 4 Reference. The Pragmatic Programmer (2013)
4. Lopes, C.V., Kiczales, G.: Improving design and source code modularity using AspectJ (tutorial session). In: ICSE, p. 825 (2000)