Access Control Based on Code Identity for Open Distributed Systems

Andrew Cirillo and James Riely

CTI, DePaul University
{acirillo,jriely}@cs.depaul.edu

Abstract. In computing systems, trust is an expectation on the dynamic behavior of an agent; static analysis is a collection of techniques for establishing static bounds on the dynamic behavior of an agent. We study the relationship between code identity, static analysis and trust in open distributed systems. Our primary result is a robust safety theorem expressed in terms of a distributed higher-order pi-calculus with code identity and a primitive for remote attestation; types in the language make use of a rich specification language for access control policies.

Keywords: Trusted Computing, Remote Attestation, Access Control, Authorization Logic, Compound Principals, Higher-Order Pi Calculus, Typing.

1 Introduction

Trust is an important concept in computer security. One may think of trust as an expectation on the behavior of some agent. We say that an agent is trusted if the achievement of a security goal is dependent on the agent behaving in the expected way. An agent is trustworthy if it behaves in the expected way in all circumstances.

An effective way to determine that an agent is trustworthy is to establish bounds on its behavior through static analysis of its software components. Many important security-related behavioral properties can be usefully established statically, including memory and type safety, non-interference, compliance with mandatory and discretionary access control policies and adherence to an ad-hoc logical policy specification.

An open system is one in which software components are under the control of multiple parties whose interests do not necessarily coincide. The use of static analysis in these systems is more complicated than in closed systems, where all components are under the control of a single party.

To discuss the issues involved, we find it useful to distinguish software components according to their relative roles. Given a particular unit of code and a statically derivable property, we distinguish four primary roles: the producer is the original author of the code; a host is a system that executes, or is considering executing, the code; a certifier is a third party capable of performing an analysis directly on the code that determines whether the property holds; and a relying party is the entity whose safe operation depends on the property holding for the code.

* This work was supported by the National Science Foundation under Grant No. 0347542.
When code is distributed in a compiled format, it may be the case that only the producer, who has the original source, is able to tractably certify many important properties. A host for the compiled code, if it is a relying party, may not be able to establish the properties it needs.

This problem is well studied, and at least two solutions have been developed. By distributing the executable as intermediate-level bytecode, the analysis may be made tractable; in this case many useful analyses may remain intractable, or at least impractical. With proof-carrying code [1] the producer uses a certifying compiler to generate a proof of the desired property that can be checked efficiently by the host; this allows a greater range of analyses, but with the limitation that properties have to be agreed upon in advance.

A second issue arises when the relying party and host systems are physically distinct. For example, a server may hold sensitive data that it is only willing to release to remote clients that are known to be running certifiably safe code. The certification could be done by the client, but on what grounds can the server trust the results? The certification can instead be done by the server, but only if it can authenticate the code running on the client.

In conventional authentication protocols, remote parties authenticate themselves by demonstrating knowledge of a secret. When executables are distributed over public channels, however, embedded secrets are vulnerable to extraction and misuse by attackers so code cannot in general be relied upon to authenticate itself. This problem is addressed in part by trusted computing, where a trusted host authenticates the code it is running, and when necessary attests to the identity of the code to remote parties.

Remote code authentication, or attestation, is based on measurements of static executables. Therefore, trusted computing platforms only attest to initial states of processes. This makes static analysis particularly important for reasoning in systems using attestation. Code identity is a degenerate example of a static property; more abstract properties can be defined as sets of executables that satisfy the property. Knowing that the executable running on a host satisfies a certain property may allow a relying party to determine something about the dynamic state of the host.

Even weak static properties may be useful in validating trust. For example, knowing that a server has the latest patches applied may ease the mind of an e-commerce client. Similarly, a bounded model checker or test suite may give some assurance of memory safety without proving absolute trustworthiness.

For concreteness, we concentrate here on access control properties established via a type system, leaving the general case to future work. This focus allows us to establish absolute guarantees of trustworthiness and thus to prove a robust safety theorem. We do so in the context of a higher-order π-calculus enhanced with process identity and primitive operations for remote attestation.

The contributions of this paper are twofold. First, we illustrate how the trusted computing paradigm can be used to enforce an access control model based on static properties of code. Second, we demonstrate the importance of higher-order languages in studying policies and protocols that make use of remote attestation.

Organization. In the remainder of this introduction, we provide some intuitions about our formalism and results. In Section 2 we present the syntax and operational semantics of our language. Detailed examples follow in Section 3. Section 4 summarizes the type