TrustBuilder2: A Reconfigurable Framework for Trust Negotiation

Adam J. Lee[1], Marianne Winslett[2], and Kenneth J. Perano[3]

[1] Department of Computer Science, University of Pittsburgh
adamlee@cs.pitt.edu
[2] Department of Computer Science, University of Illinois at Urbana-Champaign
winslett@cs.uiuc.edu
[3] Sandia National Laboratories
perano@ca.sandia.gov

Abstract. To date, research in trust negotiation has focused mainly on the theoretical aspects of the trust negotiation process, and the development of proof of concept implementations. These theoretical works and proofs of concept have been quite successful from a research perspective, and thus researchers must now begin to address the systems constraints that act as barriers to the deployment of these systems. To this end, we present TrustBuilder2, a fully-configurable and extensible framework for prototyping and evaluating trust negotiation systems. TrustBuilder2 leverages a plug-in based architecture, extensible data type hierarchy, and flexible communication protocol to provide a framework within which numerous trust negotiation protocols and system configurations can be quantitatively analyzed. In this paper, we discuss the design and implementation of TrustBuilder2, study its performance, examine the costs associated with flexible authorization systems, and leverage this knowledge to identify potential topics for future research, as well as a novel method for attacking trust negotiation systems.

1 Introduction

Recent research in trust negotiation has been primarily of a theoretical nature, focusing on a number of important issues including languages for expressing resource access policies (e.g., [1, 2, 8, 18]), protocols and strategies for conducting trust negotiations (e.g., [3, 12, 13, 28]), and logics for reasoning about the outcomes of these negotiations (e.g., [4, 27]). These results provide a strong theoretical foundation upon which provably-secure authorization systems can be designed, built, and verified. Some of the techniques discussed in the trust negotiation literature have also been shown to be viable solutions for real-world systems through a series of implementations (such as those presented in [3, 9, 11, 26]) that demonstrate the *feasibility* of using these theoretical advances. However, after several years of research, trust negotiation protocols have yet to make their way into the mainstream.

Prior to deploying access control systems based on trust negotiation, the systems and architectural properties of this technique must be more fully understood. Existing trust negotiation implementations have been developed largely as proofs of concept designed to illustrate the feasibility of the underlying theory and have performed admirably in this capacity. Unfortunately, these proof-of-concept implementations can be difficult to configure and use, and are generally not easily extended or modified. As a result, exploring certain types of systems research problems surrounding trust negotiation becomes difficult. For example:

- Is it possible to unify the myriad formulations of trust negotiation described in the research literature under a common *framework*? Adopting such a framework would make it possible to further deploy and experiment with novel trust negotiation systems and components in a grassroots fashion.
- What are the performance bottlenecks of the trust negotiation *process*, as opposed to those of a specific *implementation*? How can we quantify these costs?
- How can we identify and measure the severity of attacks that are made possible by various approaches to trust negotiation?
- When all other factors are held constant, what are the costs and benefits of using one trust negotiation system component (e.g., negotiation strategy, policy compliance checker, etc.) over another? To what extent do various approaches limit or mitigate attacks?

In an effort to address these types of systems research challenges, we have developed TrustBuilder2, a flexible and reconfigurable Java-based framework for supporting trust negotiation research.[4] TrustBuilder2 supports a plug-in based architecture to allow *any* system component to be modified or replaced by users of the system without requiring modification or recompilation of the underlying framework. TrustBuilder2 is also agnostic with respect to the formats of credentials and policies used during the negotiation. Support for new policy languages, credential formats, or trust negotiation evidence types (e.g., trust tickets [3], uncertified claims [3, 4], or proof fragments [27]) can be incorporated by implementing extensions to the TrustBuilder2 data type hierarchy. In this paper, we discuss the design and implementation of TrustBuilder2, as well the results of research carried out using this framework. Specifically, we make the following contributions:

- TrustBuilder2 represents the first fully-configurable framework for trust negotiation. TrustBuilder2 leverages a plug-in based architecture, extensible data type hierarchy, and flexible communication protocol to provide a framework within which numerous trust negotiation protocols and system configurations can be quantitatively analyzed.
- Studies carried out using TrustBuilder2 have identified the primary performance bottlenecks of the trust negotiation process. This has led to the

[4] TrustBuilder2 can be downloaded from `http://dais.cs.uiuc.edu/tn`.

identification of a novel class of denial of service attacks against trust negotiation systems that differs significantly from the attacks discussed in the research literature.
- TrustBuilder2 demonstrates that adding a high degree of flexibility to advanced authorization frameworks does not necessarily need to incur high overheads. In Section 6, we show that the time spent handling the indirection needed to support user plug-ins and other extensions amounts to less than 0.2% of the total execution time.

The remainder of this paper is organized as follows. In Section 2, we examine previously-developed trust negotiation implementations and discuss the features provided by these systems. In Section 3, we identify a number of useful features that should be provided by frameworks designed to facilitate research on the systems aspects of trust negotiation and the eventual deployment of authorization systems based on trust negotiation; we then identify the subsets of these desiderata that are addressed by existing trust negotiation implementations. Section 4 presents the architecture of the TrustBuilder2 framework for trust negotiation. In Section 5, we explore the ways in which TrustBuilder2 can be extended. Section 6 discusses a performance evaluation of the TrustBuilder2 framework and lessons learned through this process. In Section 7, we examine how TrustBuilder2 addresses the desiderata presented in Section 3. We also discuss attacks on trust negotiation systems, potential research topics uncovered by our performance evaluation, and describe how to obtain the TrustBuilder2 framework. We then present our conclusions in Section 8.

2 Background and Related Work

Trust negotiation [25] has been proposed as a potential solution to the recognized problems associated with performing access control in open systems. In trust negotiation, the access policy for a resource is written as a declarative specification of the attributes that an authorized entity must possess to access the resource. In these systems, digital credentials are issued by trusted parties to certify user attributes. For example, a student might have a digital student ID card issued by her university. These credentials are also considered resources, so sensitive credentials can be protected by disclosure policies of their own. In this way, an access request leads to a bilateral and iterative disclosure of credentials and policies between the user and resource provider. Trust is established incrementally, as more and more sensitive credentials are disclosed between the user and resource provider.

Over the last several years, several implementations of trust negotiation systems have been described in the literature. The earliest such implementation was the TrustBuilder architecture for trust negotiation [26]. TrustBuilder is a Java implementation that supports the use of X.509 certificates to encode attributes and XML to represent policies written using the IBM Trust Policy Language (TPL) [8]. The IBM Trust Establishment (TE) compliance checker is used to determine whether a certain set of credentials satisfies a given policy. TrustBuilder

has been embedded into an implementation of TLS [9] and several other protocols to demonstrate the applicability of trust negotiation in existing systems. Unfortunately, TrustBuilder supports the use of only one credential format, one policy language, and one trust negotiation strategy.

Trust-$\mathcal{X}$ [3] is an XML-based framework for supporting trust negotiations in peer-to-peer systems. In Trust-$\mathcal{X}$, each user creates an $\mathcal{X}$-profile that stores $\mathcal{X}$-TNL certificates describing their attributes along with uncertified declarations containing information about the user (e.g., preferences, phone numbers, or other such information). To the best of our knowledge, Trust-$\mathcal{X}$ does not support credential formats other than $\mathcal{X}$-TNL certificates nor policies specified in any language other than $\mathcal{X}$-TNL. To allow users to optimize various aspects of the trust negotiation process, Trust-$\mathcal{X}$ supports a variety of interchangeable trust negotiation strategies. Another particularly innovative feature of the Trust-$\mathcal{X}$ framework is its support for *trust tickets*. Trust tickets are receipts that attest to the fact that a user recently completed some negotiation with another party. These trust tickets can then be presented within some limited lifetime (typically 24-48 hours) to bypass redundant portions of future negotiations with the same party.

In [11], Koshutanski and Massacci describe a trust negotiation framework designed for web services. This framework facilitates the composition of access policies across the constituent pieces of a workflow, the discovery of credentials needed to satisfy these policies, the management of the distributed access control process, and the logic to determine what missing credentials must be located and provided to satisfy a given policy. The use of X.509 and SAML credentials is supported by the framework, as is the use of the negotiation strategies described in [11] and [13]. Policies are represented using a Datalog-based language. To the best of our knowledge, the use of other credential formats, negotiation strategies, or policy languages is not supported.

In [7], De Coi and Olmedilla describe a flexible and expressive trust negotiation implementation. The authors examined the PeerTrust [20] and Protune [5] systems in an effort to derive a set of common requirements that should be supported by any trust negotiation implementation, and then implemented a framework embodying these requirements. Their system supports PeerTrust and Protune inference engines, and allows users to add support for other inference engines. Furthermore, users can specify trust negotiation strategies as *action selection algorithms* within their framework. Credentials are expressed as signed logical statements and are loaded from a credential repository that is accessed by their implementation. To the best of our knowledge, the use of other credential formats is not supported.

While not specifically an implementation of a trust negotiation framework, Cassandra [1] is a policy language for distributed access control that supports the specification of policies with a tunable level of expressiveness. The features of Cassandra are such that it can encode a certain, fixed, trust negotiation strategy. A prototype system that uses the Cassandra language has been implemented in OCaml to facilitate research on the features of this policy language.

3 System Requirements

Prior to designing the TrustBuilder2 framework for trust negotiation, we first sought to identify the types of features that should be provided by such a framework. To this end, we studied potential uses of trust negotiation in the realms of client/server interactions on the World Wide Web, grid computing, and decentralized information sharing in critical infrastructures. Although space limitations prohibit a full treatment of these use cases,[5] the requirements identified merit discussion since they directed the design of TrustBuilder2. The first set of requirements that we identified relate to the general functionality afforded by the core components of the trust negotiation system.

Arbitrary Policy Languages. In many cases, resource providers will wish to be accessible to as many potential clients as possible. To facilitate this, these entities should be able to parse access policies written in a variety of formats (e.g., Cassandra [1], $\mathcal{X}$-TNL [2], TPL [8], RT [18], and XACML [19]). It should possible to add support for new policy languages to deployed systems easily.

Arbitrary Credential Formats. To further enable interactions with a maximal set of users, the system should support the use of multiple credential formats such as X.509 certificates [10] and SAML assertions [6]. It should also be possible to add support for new credential formats to deployed systems easily.

Interchangeable Negotiation Strategies. Trust negotiation is by nature a strategy-driven process. Entities should be able to choose negotiation strategies that direct the execution of a trust negotiation session to meet their particular goals (e.g., maximizing privacy or minimizing latency). One can imagine many situations in which the goals of the participants in a negotiation might be conflicting. The use of families of interoperable strategies that allow negotiation participants to choose different, yet compatible, strategies (e.g., as in [28]) should be supported. It should be possible to add support for new negotiation strategies to deployed systems.

Flexible Policy and Credential Stores. Clients are likely to utilize several computing devices—such as desktop computers, laptops, PDAs, and smart phones—during the course of their daily activities. It is therefore important that a trust negotiation architecture support interactions with a variety of flexible policy and credential stores (e.g., [21, 24]) that will enable users to effectively manage their digital identities across multiple devices.

While these basic flexibility requirements are important, they do not address all aspects of the negotiation process. In particular, we must also consider the ability to add more advanced features that might increase the efficiency, understandability, or functionality of the trust negotiation process.

Strategy-Driven External Interactions. Negotiation participants should have the ability to interact with a wide range of external entities that can help

[5] The complete details of our use case analysis can be found in [14].

solve difficult problems which may arise during the negotiation. Examples of such interactions might include the calculation of reputations or credential chain discovery. These interactions should be strategy-driven to allow participants to control the amount of time and resources spent pursuing these interactions.

Advanced Logging Capabilities. The architecture should include a logging service that can record information regarding any aspect of the negotiation process. Since a high degree of logging is not always needed, the logging subsystem should support the recording of logs at various granularities.

Tunable Human Involvement. In some instances, humans may wish to be involved directly in the negotiation process. For example, users may want to specify an "ask me" release policy for a sensitive credential, see a visual representation of the negotiation process for policy evaluation purposes, or be involved in the decision-making process when the negotiation comes to a point where there are multiple execution paths that could be followed rather than relying on a predefined strategy. The framework should support extensions that can add a human "in the loop" if such features are requested.

Selective Feature Activation. To enable more efficient or more secure trust negotiation sessions, the features enabled by the framework should be fully configurable. For instance, disabling support for visualization features and external interactions might increase the performance of the system, while disabling third-party plug-ins might increase overall system security and trustworthiness.

Feature Ordering. To enhance the performance of the system and its robustness against attack, entities should have the ability to choose the order in which certain functionalities are invoked. For instance, it should be possible for a negotiation strategy to choose the time at which credentials are validated. That is, there may be benefits to delaying credential validation until it is determined that they belong to a minimal satisfying set for some policy, rather than validating them as they are received.

The diversity of use cases that we considered leads us to believe that it represents a useful set of features to support when designing a general-purpose trust negotiation framework. However, the requirements presented above cannot be considered complete, as it is impossible to consider every possible trust negotiation use case. To acknowledge and partially address this gap, we introduce one further requirement that helps ensure additional features can be easily added to the framework.

Extensibility. The framework must support the addition of new functionality after deployment without requiring modifications to the existing code base. Example features may include (but are not limited to) the inclusion of new local data processing rules, the enforcement of obligations, and the incorporation of new data types into the negotiation process.

Table 1 identifies the subsets of these requirements addressed by each of the trust negotiation frameworks discussed in Section 2. As shown, no existing trust

	TrustBuilder	Trust-$\mathcal{X}$	Koshutanski	De Coi	Cassandra
Arbitrary policy languages	N	N	N	P	P
Arbitrary credential formats	N	N	P	N	N
Interchangeable negotiation strategies	N	P	P	Y	N
Flexible policy and credential stores	N	N	N	N	N
External interactions	N	N	N	Y	Y
Tunable human involvement	N	N	N	N	N
Advanced logging	N	N	N	P	N
Selective feature activation	N	N	N	N	N
Feature ordering	N	N	N	N	N
Extensibility	N	N	N	P	P

Table 1. Features supported by existing trust negotiation implementations (Y = yes, N = no, P = partially supported).

negotiation framework provides even partial support for more than half of the identified features; this is not surprising, given that these implementations were not meant to be general-purpose frameworks.

4 The TrustBuilder2 Framework

In this section, we describe the design of TrustBuilder2, a Java-based framework for trust negotiation. The primary goal in designing TrustBuilder2 was not to implement one particular trust negotiation protocol, but rather to provide a framework that satisfies the requirements set forth in Section 3, within which any number of trust negotiation techniques can be implemented and evaluated. This led to unique challenges in designing the communication protocol used by negotiation participants, the data type hierarchy used by TrustBuilder2, and the software architecture of the system. In this section, we describe the above facets of the TrustBuilder2 framework.

4.1 Communication Protocol and Data Types

One of the first challenges faced when designing TrustBuilder2 was defining a communication protocol that could be interpreted by the framework without constraining the trust negotiation protocols that could be supported. For example, we did not want to mandate that *only* credentials and policies are exchanged during a trust negotiation session, as that would prevent the implementation of protocols such as Trust-$\mathcal{X}$ [3] and PeerAccess [27] within the TrustBuilder2 framework, since these protocols also exchange digitally-signed trust tickets and proof-fragments, respectively. To this end, TrustBuilder2 uses a very simple communication protocol combined with an extensible data type hierarchy to enable the implementation of a wide range of trust negotiation protocols.

Data type hierarchy. At a high level, a trust negotiation session is an exchange of messages containing credentials, policies, uncertified claims, and other information between two parties. In order to support the widest possible range of

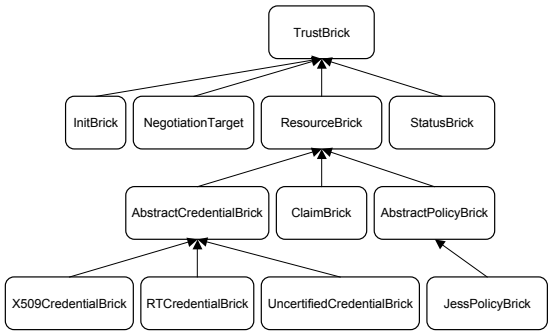

Fig. 1. Class hierarchy for several important TrustBrick subclasses.

trust negotiation protocols, the core components of the TrustBuilder2 framework (described in Section 4.2) rely heavily on the use of an extensible data type hierarchy. All types of information that might be exchanged between negotiating parties are represented as subclasses of the TrustBrick class, which forms the basic building block of the trust negotiation process. In this way, users can extend the data types supported by TrustBuilder2 without modifying each component in the system; components can simply ignore TrustBricks that they do not know how to process, leaving them for other system components to handle. Entities then exchange TrustMessage objects containing one or more of these TrustBricks.

Figure 1 shows the relationships between several important subclasses of TrustBrick. The InitBrick, NegotiationTarget, and StatusBrick classes are used to provide high-level information regarding a negotiation: InitBricks are used to establish the parameters of a trust negotiation, a NegotiationTarget is used to indicate the particular resource that the initiator of a trust negotiation wishes to access, and a StatusBrick may be included in the last message of the negotiation to indicate whether or not the negotiation succeeded in establishing trust between the participants. Any item exchanged during a trust negotiation that could possibly be protected by a release policy is a subclass of ResourceBrick. This ensures that TrustBuilder2 can properly enforce disclosure requirements on data items without necessarily understanding the data item itself.

The AbstractCredentialBrick and AbstractPolicyBrick classes are used to represent attribute certificates and policies at an abstract level, which enables components of TrustBuilder2 to handle credentials and policies of various formats without needing to understand the intricacies of each format explicitly. The X509CredentialBrick class is used to hold information about X.509 certificates, while the RTCredentialBrick class holds information about *RT* credentials [18]. The UncertifiedCredentialBrick class provides TrustBuilder2 with the ability to create "fake" credentials on-the-fly to facilitate the rigorous testing of system components as they are developed. Lastly, the JessPolicyBrick class is used to hold policies that can be interpreted by the CLOUSEAU compliance checker [15].

In Section 5, we illustrate the ways in which this extensible type hierarchy facilitates the extension of the TrustBuilder2 framework to incorporate new features, such as support for new policy languages or credential types. Readers interested in more detail regarding TrustBrick or its subclasses should consult the TrustBuilder2 programmer documentation included with the TrustBuilder2 distribution.

The communication protocol. As previously mentioned, the TrustBuilder2 communication protocol is nothing more than an exchange of TrustMessage objects containing one or more TrustBricks between the participants of the negotiation. The first message sent by the initiator of the trust negotiation session contains a single InitBrick object describing the TrustBuilder2 system configurations (i.e., strategy families, credential formats, and policy languages) that she supports, along with other system parameters. If the responder supports a system configuration that is compatible with one of the system configurations proposed by the initiator, he returns a TrustMessage containing another InitBrick describing this system configuration. At this point, both parties can configure their TrustBuilder2 framework to use this compatible system configuration during their negotiation session. The initiator then responds with a TrustMessage containing a NegotiationTarget that indicates the resource that she wishes to access. Beyond this, no constraints are imposed on the contents of these messages; future TrustMessage objects exchanged by the participants are handled by the strategy modules (described in the next section) supported by each of the participants, rather than the core TrustBuilder2 framework. This allows TrustBuilder2 to support a wide range of trust negotiation protocols without requiring protocol-specific modifications be made to the framework itself.

4.2 Software Architecture

Figure 2 presents a high-level architecture diagram of the TrustBuilder2 runtime system. Note that components enclosed in dashed boxes are not included in the current version of TrustBuilder2; they are only meant to serve as example components that could be developed by users as plug-ins and added to the TrustBuilder2 data path. We now describe each of the major components identified in this diagram and comment on the flow of data between components.

The external interface to the TrustBuilder2 runtime system is provided by the TrustBuilder2 class. Trust negotiation sessions are conducted by making a series of calls to methods exposed by this class. When a new trust negotiation session is started, the TrustBuilder2 class creates and manages a Session object that keeps track of all necessary state between rounds of the negotiation. For example, after the exchange of InitBricks described in Section 4.1, the Session object will contain a description of the TrustBuilder2 configuration to be used for this session, including the strategy module to use, a list of supported policy languages, and a list of supported credential formats. Any component in the system can add its own internal state to a given Session object. This allows

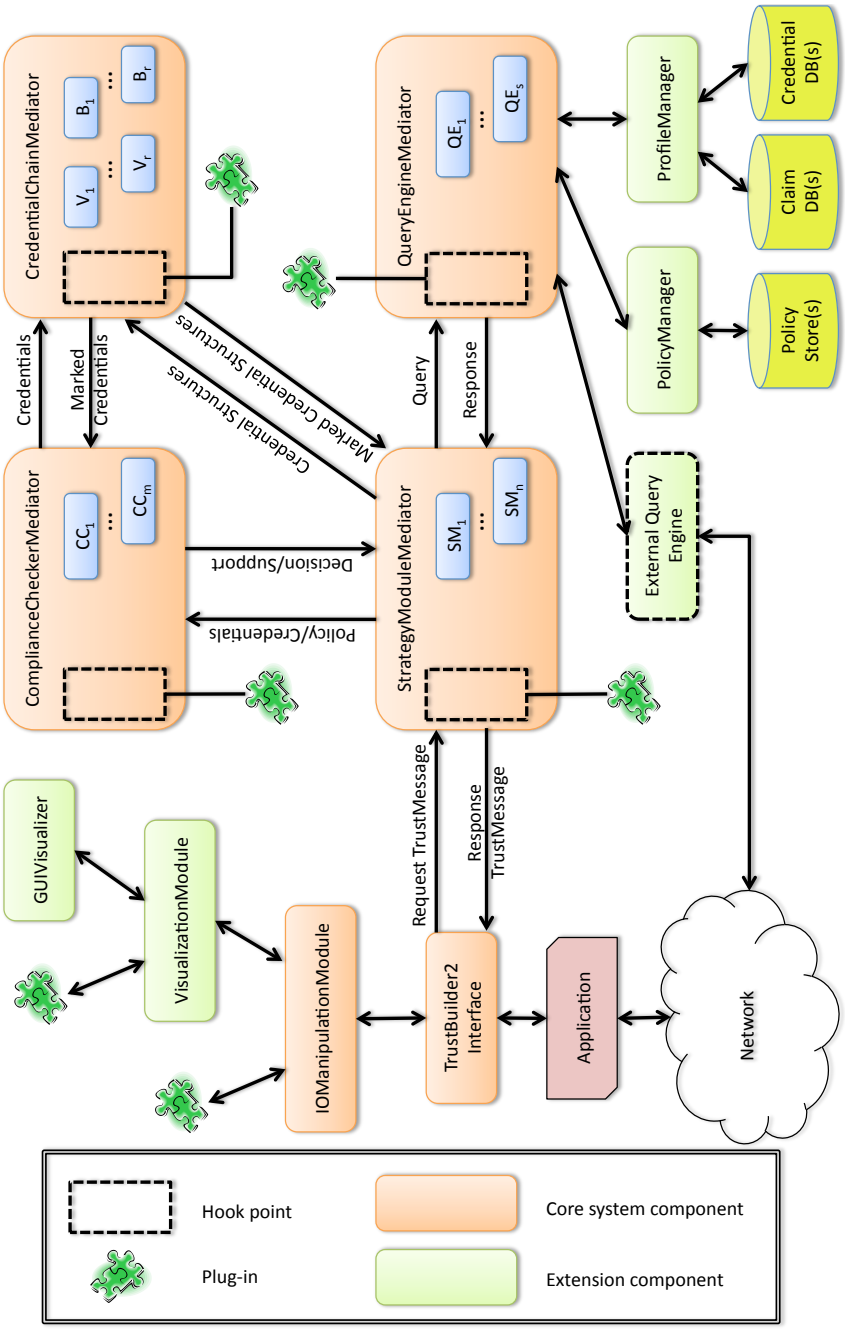

Fig. 2. TrustBuilder2 architecture overview diagram.

components to avoid maintaining this state locally and eases the development of reentrant system components.

During the trust negotiation process, all incoming TrustMessages are processed by the TrustBuilder2 object, which generates a response TrustMessage to return to the remote participant. Prior to processing a remote TrustMessage itself or dispatching it to the StrategyModuleMediator, the TrustBuilder2 object first passes all incoming messages to the IOManipulationModule. The IOManipulationModule is the first class to process each incoming TrustMessage and the last class to process each outgoing TrustMessage. This component is capable of loading user-defined plug-ins that can examine and modify all TrustMessage objects entering and leaving the TrustBuilder2 runtime system. The VisualizationModule is an example plug-in to the IOManipulationModule that provides an interface for writing and using custom logging and visualization components. The TrustBuilder2 distribution includes two such components: the GuiVisualizer class is a plug-in that uses the Swing API to graphically visualize every TrustMessage processed by TrustBuilder2, while the BasicConsoleVizualizer is a plug-in that provides a console logging facility.

The core of the TrustBuilder2 runtime system—that is, the interfaces to the strategy modules, compliance checkers, credential and policy stores, and credential manipulation routines—is provided by a set of four *mediator* classes. Each mediator class acts as a dispatcher providing access to any number of user-specified trust negotiation system core components. The StrategyModuleMediator is responsible for managing the set of installed trust negotiation strategies. Strategies encode the "brains" of a trust negotiation session; given an incoming TrustMessage and the existing negotiation state, a strategy responsible for interacting with the ComplianceCheckerMediator to analyze policies, the CredentialChainMediator to construct and verify credential chains, and the QueryEngineMediator to access local trust negotiation evidence or interact with external query services. It then uses the information gleaned from this process to generate a response TrustMessage that will be sent to the remote party. Each mediator class provides *hook points* that allow user-developed plug-ins to intercept all calls into the mediator class and all returns from the mediator class. This provides an easy way for users to monitor or modify the flow of information through the TrustBuilder2 framework.

For the sake of brevity, not every component of TrustBuilder2 was discussed in this section. Readers desiring a more complete treatment of the components of the TrustBuilder2 system should consult the programmer documentation available in the TrustBuilder2 distribution.

4.3 Default Configuration and Extensibility

By default, TrustBuilder2 includes support for a version of the TrustBuilder1-Relevant strategy for trust negotiation described in [28] modified to further minimize information disclosure in the event that multiple satisfying sets are found for a given policy during a negotiation. As described above, TrustBuilder2 supports the use of X.509 V3 credentials during interactions with remote parties but

can also use uncertified "test" credentials to exercise the functionality of new plug-ins or components as they are being designed and developed. Plug-ins are provided for the CredentialChainMediator that allow TrustBuilder2 to form a set of credential chains from a collection of credentials of *any* format and to verify the authenticity of the credential chains that were formed. TrustBuilder2 currently supports the CLOUSEAU compliance checker and can load a user's policies, credentials, and uncertified claims from repositories on the local file system.

5 Case Studies in Extensibility

In this section we discuss the extensibility of TrustBuilder2 at a high level, as well as provide a more detailed treatment of two significant extensions added to the framework after its initial development.

5.1 General Extensibility

As was our goal from the outset, almost every component of the TrustBuilder2 framework can either be extended or replaced by a user-defined plug-in. Because the TrustBuilder2 framework was developed using Java, dynamic class loading can be used to incorporate these user plug-ins at runtime without requiring any modification to the TrustBuilder2 framework itself. Extensions to the primary components of TrustBuilder2—that is, the IOManipulationModule, StrategyModuleMediator, ComplianceCheckerMediator, CredentialChainMediator, and the QueryEngineMediator—as well as plug-ins that interpose between these components, can be added the system quite easily. Users simply write and compile plug-ins conforming to the appropriate interfaces and instruct the TrustBuilder2 runtime system to incorporate these modules the next time that a TrustBuilder2 object is created. For instance, adding a new strategy to TrustBuilder2 involves writing a class implementing StrategyModuleInterface and adding this class to the list of strategy modules to be loaded by the StrategyModuleMediator.

We now discuss how the abstract type hierarchy used by TrustBuilder2 allows support for new credential and policy formats—as well as new forms of negotiation evidence—to be added to the the system without requiring modifications to the underlying framework. As will be shown, this process is very straightforward and allows support for novel trust negotiation features to be easily incorporated into the TrustBuilder2 framework.

5.2 X.509 Credentials and Uncertified Claims

Initially, the TrustBuilder2 framework only included support for uncertified "test" credentials, as encoded by the UncertifiedCredentialBrick class. These credentials can be easily created and modified and thus allow for rapid and efficient testing of system components. However, to better study the properties of trust negotiation systems that might be deployed in practice, support for more realistic credential types was required. As a result, we added support for uncertified

claims encoding user data such as phone numbers or preferences (as in [3, 4]), as well as X.509 v3 certificates to the TrustBuilder2 framework.

Supporting uncertified claims required the following two extensions be made to TrustBuilder2. First, the ClaimBrick data type was added as a subtype of the ResourceBrick data type in the TrustBuilder2 type hierarchy (see Figure 1). Subtyping ResourceBrick in this way ensures that uncertified claims can be treated as sensitive and optionally protected by release policies. Second, a loader plug-in was written for the ProfileManager so that uncertified claims could be loaded from the file system. In total, less than 300 lines of commented code had to be written to support the addition of uncertified claims to TrustBuilder2.

Adding support for X.509 v3 certificates to TrustBuilder2 was accomplished in a similar manner. Specifically, the X509CredentialBrick data type was added as a subtype of the AbstractCredentialBrick type and another loader plug-in was written for the ProfileManager so that X.509 certificates could be loaded from the file system. The X509CredentialBrick data type wraps the functionality of the X.509 data type supported by Java natively and provides additional methods that extract attribute information from a credential's extension OID fields, populate the data structures used by AbstractCredentialBrick objects, create and verify proof of ownership challenges, and verify the issuer signatures. Since TrustBuilder2's default policy compliance checker, credential chain construction algorithms, and credential chain verification algorithms operate on AbstractCredentialBrick objects, no further modifications were needed for TrustBuilder2 to support X.509 v3 certificates. Fewer than 1000 lines of commented code were needed to implement the plug-ins required to include this support.

5.3 RT Credentials and Policies

To further extend the functionality of TrustBuilder2, we have also implemented support for RT_0 and RT_1 credentials and policies [18]. Adding support for the necessary credential types involved a process similar to that followed for supporting X.509 credentials. That is, TrustBuilder2's type hierarchy was extended to include RT credentials, and a loader plug-in was written to read these credentials from disk. However, since Java does not support RT credentials natively, considerably more code had to be written than was the case for adding support for X.509. In total, approximately 3500 lines of commented code were required to add support for loading, parsing, and using these types of credentials within the TrustBuilder2 framework.

In general, adding support for a new policy language to TrustBuilder2 would require developing a new policy compliance checker that is capable of analyzing the satisfaction of this new type of policy. Such a compliance checker would take the form of a plug-in to the ComplianceCheckerMediator. However, this was not the case for RT_0 and RT_1 policies. Recent results [15] show that these types of policies can actually be compiled into a format that can be efficiently analyzed by the Clouseau compliance checker, which is already supported by TrustBuilder2. Currently, policies must be compiled in an offline manner prior to being used

by TrustBuilder2, which limits the credential chain discovery functionality supported by *RT*. To overcome this barrier, we plan to implement a plug-in that interposes between the StrategyModuleMediator and the ComplianceCheckerMediator and compiles *RT* policies at runtime.

6 Performance Evaluation and System Profiling

We now discuss the results of a performance evaluation of the TrustBuilder2 framework. Our primary goals in this investigation were to evaluate the overheads associated with the flexible nature of TrustBuilder2 and to better understand the bottlenecks involved in the trust negotiation process. We then discuss a novel type of denial of service attack and several potential research directions uncovered during this analysis.

6.1 The Scenario

Our scenario was designed to mimic a trust negotiation scenario that might take place in one branch (Acme Springfield) of a national-scale corporation (Acme Fabrication). In this scenario, an employee wants to access a file server containing sensitive files related to "Project X." The policy protecting the Project X file repository states that an authorized entity must be either (i) a full-time employee of Acme Springfield that has an Acme Fabrication issued sensitive document training certification and works in department 2460–2469 or (ii) a full-time employee of Acme Springfield that has an Acme Fabrication issued sensitive document training certification, works in department 2400–2499 and was granted an "access exception" for Project X by either Alice or Bob. This policy was thought to be a reasonable example of a negotiation that one might see in a large corporation as it is much simpler than managing a long access control list, but also includes provisions for the explicit white-listing of people who are not authorized by the blanket policy. Furthermore, entities on the white-list can easily be traced back to the employee authorizing them

The client in our scenario has a valid employee ID stating that he is a full-time employee in department 2442 of Acme Springfield, a sensitive documents training credential, and an access exception issued by Bob. Figure 3 illustrates this example negotiation scenario graphically. The first two messages exchanged during the negotiation contain configuration information used by TrustBuilder2 to establish the parameters for the negotiation session. The second message sent by the client indicates his interest in accessing the file server associated with Project X. The second message sent by the file server releases the policy protecting this file server to the client. The client can satisfy this policy, but is not willing to disclose his security clearance or access exception unless the server can prove that it is operated by Acme Springfield. As such, the third message sent by the client discloses the release policy protecting these credentials and the credential chain ending with his employee ID. Note that supporting credentials are not shown in Figure 3. At this point, the file server validates the

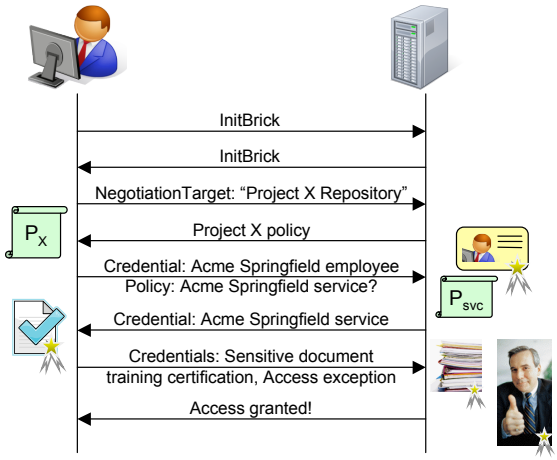

Fig. 3. A simplified view of the trust negotiation used during our experiments.

proof-of-ownership associated with the employee's employee ID and accepts this credential. It also then discloses the credential chain that identifies the file service as operated by Acme Springfield. The client verifies this credential chain and the proof-of-ownership associated with the leaf credential in the chain and then discloses his sensitive documents training credential and his access exception to the file server. The file server verifies the proofs-of-ownership associated with these credentials and then grants the client access to the service.

6.2 The Experiments

We used the above trust negotiation scenario to conduct two experiments. In the first experiment, a client application made a TCP connection to a server application and carried out the trust negotiation described by Figure 3 using an ObjectOutputStream to write TrustMessages to the remote server and an ObjectInputStream to read response TrustMessages. When the negotiation succeeded, the client would disconnect from the server. This entire process was repeated 100 times. The client and server applications were both executed from the system command prompt using JDK 1.5.0_06. This experiment was designed to enable us to study the average execution time of a trust negotiation session.

In the second experiment, we sought to profile the execution of the TrustBuilder2 runtime system to gain a better understanding of the costs of the various components of a trust negotiation. In this experiment, the server process described above was started from the system command line using JDK 1.5.0_06. The client application was loaded into the Eclipse development environment and profiled using the Eclipse Test and Performance Tools Platform (TPTP) tracing and profiling tools plug-in version 4.2.1.

In our experiments, the TrustBuilder2 objects used by the client and server processes supported only the use of X.509 credentials encoded as

X509CredentialBrick objects. All X.509 credentials used during this scenario encoded RSA key pairs. Further, each credential was represented as a unique X.509 certificate with its own key pair. Both the client and server processes supported the use of the Clouseau compliance checker. The strategy used by both parties was the variant of the TrustBuilder1-relevant strategy discussed in Section 4.3 that is implemented by the MaximumRelevantStrategy class included in the TrustBuilder2 distribution. Credential chains were built using the SimpleChainBuilder class and verified using the RootToLeafVerifier class. The IOManipulationModule was disabled at both the client and server. The experiments described above were run using a single machine, rather than two machines, as we were more interested in the computational costs of the trust negotiation than the communication latencies imposed by routing packets through an Ethernet network. The machine that we used had a 3.2 GHz Intel Pentium 4 processor, 1 GB of RAM, and was running Gentoo Linux (kernel 2.6.12).

6.3 Results

After conducting the first experiment, we found that the average time to conduct the aforementioned trust negotiation session using TrustBuilder2 was 434.73 ms with a standard deviation of 97.56 ms. This is at least an order of magnitude faster than a trust negotiation session carried out using the original TrustBuilder framework, as a similar negotiation takes seconds on average within that framework [16]. We did find that the first trust negotiation session took roughly three times as long as an average negotiation (1350 ms) due to the cost of Java initially loading the classes used by the TrustBuilder2 framework. We do not see this as a problem, however, as it is likely that TrustBuilder2 objects will be used for multiple negotiations and therefore this initial cost will quickly be amortized, as it was in our experiments.

In our second experiment, we found that the majority of the time spent in one of three tasks: using the compliance checker ($\approx 49\%$), reading from and writing to I/O streams ($\approx 15.5\%$), and signing proof-of-ownership challenges ($\approx 14.4\%$). We also found that the overheads required to support plug-in loading and interposition amount to less than 0.2% of the overall cost of the trust negotiation process. This implies that the flexibility afforded by the TrustBuilder2 framework does not, in and of itself, carry the steep overheads that we had originally anticipated. Of course, loading inefficient or otherwise expensive plug-ins could easily increase the cost of a trust negotiation.

7 Discussion

In this section, we revisit the requirements presented in Section 3 and discuss the ways in which they are met by TrustBuilder2. We then discuss potential implications of the performance results obtained in Section 6.

7.1 Requirements Redux

In Section 3, we introduced ten requirements that should be provided by frameworks for exploring the systems aspects of trust negotiation. Section 5 illustrated the ways in which plug-in extensions to TrustBuilder2 can be used to meet the *arbitrary policy languages*, *arbitrary credential formats*, *interchangeable negotiation strategies*, *flexible policy and credential stores*, and *extensibility* requirements. The plug-in interface for defining strategy modules does not place any constraints on how the strategy behaves, which enables user-defined strategy modules to meet the *tunable human involvement* and *feature ordering* requirements. The VisualizationModule plug-in to the IOManipulationModule enables advanced logging and visualization features, thus meeting the *advanced logging capabilities* requirement. Finer-grained logging can be accomplished by placing calls to the logger at the mediator hook points described in Section 4.2. The QueryModuleMediator can be used to allow the TrustBuilder2 framework to interact with processes external to the negotiation at hand simply by developing new query module plug-ins, thereby meeting the *strategy-driven external interactions* requirement. Finally, each of the plug-ins to the TrustBuilder2 system can be individually enabled or disabled, thereby meeting the *selective feature activation* requirement.

7.2 Attacks and Future Research

One striking result from the performance evaluation presented in Section 6 is that nearly half of a trust negotiation session is spent interacting with the compliance checker. During our experiments, the client process spent, on average, 226 ms interacting with the compliance checker during a single trust negotiation. The complexity of the compliance checking process has also been observed in other, independent trust negotiation implementations (e.g., see [23]). This suggests that a novel and highly-effective denial of service attack against trust negotiation-enabled services is to force the use of the remote party's compliance checker. An attacker can easily accomplish this by either placing release policies on every credential that might possibly be released to the remote party, or by sending spurious policies that the remote party *thinks* are protecting resources that could advance the state of the negotiation. Such an attack involves little overhead for the attacker, yet can consume arbitrary resources on the host being attacked.

This attack is quite different than the types of denial of service attacks on trust negotiation discussed in the research literature. To date, attacks against trust negotiation systems have focused on examining ways to exploit the credential chain construction and verification processes [17, 22]. These attacks leverage the disparity in cost between transmitting a credential chain and verifying that the chain is correctly formed to consume resources on the target system. The higher per-unit cost of policy compliance checking when compared to credential verification implies that attacking the compliance checker used by a trust negotiation system can be at least as damaging as attacking its credential chain verification process. Furthermore, malicious entities combining these two attacks

can slow the processes of analyzing both local and remote policies at the host being attacked.

Analyzing the cost breakdown of example trust negotiation scenarios not only led to the identification of this attack strategy, but also helped identify future research directions aiming to better optimize trust negotiation systems. For example, an earlier version of our performance analysis led us to explore alternate formulations of the policy compliance checking problem that would allow for more efficient policy analysis than existing theorem proving approaches. The result was the CLOUSEAU compliance checker, which leverages an efficient pattern matching approach to greatly outperform existing compliance checkers both asymptotically and in practice [15]. However, the attacks described above occur even when using this optimized compliance checker. An interesting direction of future research could be the development of trust negotiation strategies that can detect the above types of compliance checker abuses and either triage "unproductive" negotiations or seek to limit the use of the compliance checker without compromising the completeness property of the trust negotiation protocol.

8 Conclusions

In this paper, we presented TrustBuilder2, a flexible framework for investigating the systems aspects of trust negotiation. TrustBuilder2 supports the dynamic loading of new trust negotiation system components—such as strategy modules, compliance checkers, policy and credential storage devices, and logging and visualization modules—without modification to the underlying framework and features an extensible type hierarchy that allows end-users to easily add support for new credential formats and policy languages. By profiling the performance of TrustBuilder2, we found that the system has a number of desirable properties that make it ideal for researching the systems obstacles to deploying trust negotiation systems in practice. Furthermore, our performance evaluation enabled us to uncover a novel class of attacks against trust negotiation systems and led us to identify promising areas of future trust negotiation systems research.

Acknowledgments. This research was supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695, and by Sandia National Laboratories under grant number DOE SNL 541065.

References

1. Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '04)*, pages 159–168, 2004.
2. Elisa Bertino, Elana Ferrari, and Anna Cinzia Squicciarini. X -TNL: An XML-based language for trust negotiations. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '03)*, 2003.

3. Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust-X: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, July 2004.
4. Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *7th ACM Conference on Computer and Communications Security*, pages 134–143, 2000.
5. Piero A. Bonatti and Daniel Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *Proceedings of the Sixth IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 14–23, June 2005.
6. Scott Cantor, John Kemp, Rob Philpott, and Eve Maler (Editors). Assertions and protocols for the OASIS security assertion markup language (SAML V2.0). OASIS Standard, March 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.
7. Juri L. De Coi and Daniel Olmedilla. A flexible policy-driven trust negotiation model. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 450–453, November 2007.
8. Amir Herzberg, Yosi Mass, Joris Michaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, May 2000.
9. Adam Hess, Jared Jacobson, Hyrum Mills, Ryan Wamsley, Kent E. Seamons, and Bryan Smith. Advanced client/server authentication in TLS. In *Network and Distributed Systems Security Symposium*, February 2002.
10. Russell Housely, Warwick Ford, Tim Polk, and David Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. IETF Request for Comments RFC-2459, January 1999.
11. H. Koshutanski and F. Massacci. Interactive access control for web services. In *19th IFIP Information Security Conference (SEC)*, pages 151–166, August 2004.
12. H. Koshutanski and F. Massacci. An interactive trust management and negotiation scheme. In *2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 139–152, August 2004.
13. H. Koshutanski and F. Massacci. Interactive credential negotiation for stateful business processes. In *3rd International Conference on Trust Management (iTrust)*, pages 257–273, May 2005.
14. Adam J. Lee. *Towards Practical and Secure Decentralized Attribute-Based Authorization Systems*. PhD thesis, University of Illinois at Urbana-Champaign, July 2008.
15. Adam J. Lee and Marianne Winslett. Towards and efficient and language-agnostic compliance checker for trust negotiation systems. In *3rd ACM Symposium on Information, Computer, and Communication Security (ASIACCS '08)*, March 2008.
16. Adam J. Lee, Marianne Winslett, Jim Basney, and Von Welch. The Traust authorization service. *ACM Transactions on Information and System Security*, 11(1), February 2008.
17. Jiangtao Li, Ninghui Li, Xiaofeng Wang, and Ting Yu. Denial of service attacks and defenses in decentralized trust management. In *2nd International Conference on Security and Privacy in Communication Networks (SecureComm)*, August 2006.
18. Ninghui Li and John Mitchell. RT: A role-based trust-management framework. In *3rd DARPA Information Survivability Conference and Exposition*, April 2003.
19. Tim Moses. XACML 2.0 Core: eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard, February 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

20. Wolfgang Nejdl, Daniel Olmedilla, and Marianne Winslett. Peertrust: Automated trust negotiation for peers on the semantic web. In *Proceedings of the VLDB Workshop on Secure Data Management (SDM)*, volume 3178 of *Lecture Notes in Computer Science*, pages 118–132. Springer, August 2004.
21. Jason Novotny, Steven Tuecke, and Von Welch. An online credential repository for the grid: MyProxy. In *10th International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.
22. Tatyana Ryutov, Li Zhou, Clifford Neuman, Travis Leithead, and Kent E. Seamons. Adaptive trust negotiation and access control. In *10th ACM Symposium on Access Control Models and Technologies*, June 2005.
23. Bryan Smith, Kent E. Seamons, and Michael D. Jones. Responding to policies at runtime in TrustBuilder. In *5th International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, June 2004.
24. Tim W. van der Horst and Kent E. Seamons. Short paper: Thor — the hybrid online repository. In *1st IEEE International Conference on Security and Privacy for Emerging Areas in Communications Networks*, September 2005.
25. William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, January 2000.
26. Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, Nov./Dec. 2002.
27. Marianne Winslett, Charles Zhang, and Piero Andrea Bonatti. PeerAccess: A logic for distributed authorization. In *12th ACM Conference on Computer and Communications Security (CCS 2005)*, November 2005.
28. Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), February 2003.