TrustForge: Flexible Access Control for VehicleFORGE Collaborative Environment

Sponsored by Defense Advanced Research Projects Agency
Tactical Technology Office (TTO)
Program: VehicleFORGE – University of Pennsylvania
Issued by DARPA/CMO under Contract No. HR0011-11-C-0096

Final Report*

November 3, 2012

1 Introduction

Flexible manufacturing, based on the extensive use of models and model-based analysis, is promising to revolutionize the way we build equipment. The use of models can improve outcomes and reduce costs of designing and building complex machinery by thoroughly exploring the design space and selecting the approach that matches the requirements best, eliminating the need for expensive prototypes. Furthermore, the use of models can dramatically lower the barrier of entry for small companies and even individuals, who will be able to bring fresh ideas to the table. Ultimately, designs can be *collaborative design*, scores of participants would design models according to common requirements. This is the vision of the Adaptive Vehicle Make [1], a portfolio of programs run by Defense Advanced Project Agency (DARPA) with the goal of applying collaborative design, model-based engineering to the design of military vehicles. The idea is to leverage the talents of the wider world, rather a few select contractors to build next-generation military equipment. The success of collaborative design software platforms such as Linux, Firefox and Apache that use a similar model have paved the way for migrating the basic approach to military systems.

In this report, we present *TrustForge*, an autonomous, dynamic and flexible access control mechanism for collaborative design component-based systems. TrustForge is part of a larger repository, `VehicleFORGE`, that stores the various contributions made by the users. The users interact with the repository, which itself uses the TrustForge to make access control

*The views expressed are those of the TrustForge team and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

decisions, thereby allowing or disallowing users from performing certain tasks. In this regard, the repository provides TrustForge with meta-data on users and their components which is then used by it to compute the trustworthiness of the components and their contributors (*i.e.,* users). Traditional trust management approaches rely on credentials that are assigned to participants by trusted authorities. Such approaches rely on cryptography and give precise guarantees that only participants with the right credential can gain access to the repository. On the other end of the spectrum are reputation-based approaches that are based on prior interactions between the participants in the collaborative environment. In such approaches, participants report their experiences to the system that keeps track of problematic users. The quintessential reputation-based system is implemented by eBay, where buyers and sellers report on the outcome of transactions.

Both purely policy-based or reputation-based approach does not meet the needs of a development environment. Policies provide the ability to credential user and authorize them to access the system. Credentialing, or establishing the authority to contribute to the projects, is essential in collaborative design environments especially given the diversity of participants. The credentialing of users is a particularly challenging task in collaborative design environments because: (1) the identity of the participants may not be directly known to the project management, (2) the strictness of access control needs to be flexible depending upon the criticality of the project, (3) the scale of the enterprise in terms of the number of participants and projects and their components. Today's collaborative design software platforms perform much of the credentialing manually. It is therefore not surprising that credentialing in such scenarios tends to be a slow and time-consuming process, which provides considerable barrier of entry for new participants or the introduction of newer ways of building such platforms. Similarly, reputation systems, such as those used by eBay, based on direct user feedback on other users are susceptible to attacks, in which malicious users deliberately provide bad feedback on good users to drive their reputation down. At the same time, malicious users can collude with other malicious users to provide good feedback to each other, boosting reputation.

With TrustForge, we are therefore staking the hybrid approach for access control that allows us to incorporate constraints on user reputation in a access control policy. Reputation becomes one element in the array of attributes that the user has. Examples of others attributes may be the citizenship status and the number of years of relevant experience,and so on. In order to maintain reputation values, TrustForge periodically recalculates user reputations based on the history of interactions between the user and the repository. In order to protect the reputation scores from attacks based on malicious feedback, similar to the ones described above, the feedback in TrustForge is based on the objective information, rather than on subjective opinions of other users. The intuition for the feedback sources are as follows. Since we are primarily concerned with the quality of the components that a user submits to the repository, the reputation of the user is based on the reputation of the submitted components. The most objective assessment of component quality is through testing, simulation, or other analysis method. However, testing must be performed by trusted users, who are by necessity a minority of users. Therefore in a large repository, only a small fraction of components can be tested with sufficient confidence. Moreover, testing or simulation results, typically, do not give complete confidence. We supplement test results with the information about how components use each other. If component A is used by a large number of components with high reputation, it means that contributors of those components found A acceptable, which gives us additional confidence in the trustworthiness of A.

Even though this work was done in the context of the Adaptive Vehicle Make project. We will keep our discussion more general and present the conceptual aspects of our system and its implementation. The TrustForge architecture is general enough to be used with any collaborative design application, as long as the repository provides it the inputs it requires about the users and the components they create.

This report is organized as follows: Section 2 presents the TrustForge system is detail including the policy engine, data model for storage and reputation function. Section 4 then presents performance analysis results for TrustForge including our simulation setup. In Section 5 we conclude.

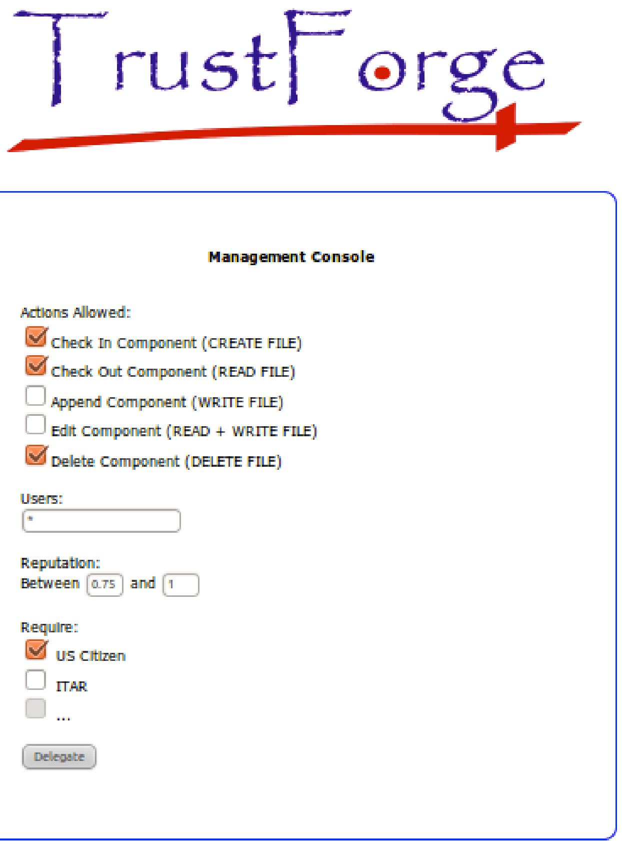
2 The TrustForge System

Given the design goals, in this section, we provide an overview of the TrustForge system architecture including the policy engine, the data model for storing user and component meta-data and relationships, the reputation function, the TrustForge API and its implementation.

2.1 Policy Specification and Evaluation

We use the KeyNote trust management language for specifying these access policies in TrustForge. KeyNote is a declarative language describing relationships among principals and evidence that permits principals to perform certain actions [2]. These relationships are specified as policies. If cryptographically signed these policies can be viewed as a credential. KeyNote credentials and policies are known as *assertions*. When a trust inquiry is made, users present a set of cryptographically signed policies (*i.e.*, credentials) along with the desired action they wish to perform. The KeyNote *compliance checker* evaluates this input and returns a Compliance Value (CV) in a linearly ordered set, between an application specified minimum compliance value and maximum compliance value. In the simplest case, the set of compliance values can be $\{\text{DENY}, \text{ALLOW}\}$. CV is then used to make the access control decision.

A KeyNote policy has three primary components; the Authorizer, Licensees, and Conditions fields. The Authorizer field is the principal delegating trust, who is issuing this assertion. KeyNote provides a special Authorizer, `POLICY`, that is the root of all trust. Valid delegation chains must emanate from `POLICY`. The Licensees field states to which principal(s) the Authorizer is delegating trust to, and it also expresses from whom delegation chains must be present. It is written as a logical statement of AND/OR operators between principals. The logic of the Licensees field also describes how CVs are combined to form a single output.



Management Console

Actions Allowed:

- [✓] Check In Component (CREATE FILE)
- [✓] Check Out Component (READ FILE)
- [ ] Append Component (WRITE FILE)
- [ ] Edit Component (READ + WRITE FILE)
- [✓] Delete Component (DELETE FILE)

Users: *

Reputation:
Between 0.75 and 1

Require:

- [✓] US Citizen
- [ ] ITAR
- [ ] ...

Figure 1: A Sample TrustForge Policy

The MAX function is applied to OR delegations and the MIN function to AND ones. Finally, the Conditions field permits comparison using context associated with the access request. Conditions are written as propositions, which when true imply a CV. When more than one value is implied, the maximal one is used as the returned CV. Provided this specification, one can now determine a CV for an entire delegation chain. First one computes CVs via the comparisons of the Conditions field for all credentials in the chain. These are then combined up the chain using MIN/MAX until the root node (`POLICY`) is reached. This final CV is then returned to the application.

The Conditions field is allows us to specify constraints on the application of a credential. A credential can be used to delegate only a specific privilege to the Licensees; for example, a credential may be allowed to check out components from the repository, but not add new ones. In this case, the Conditions field would include a predicate that compares the requested action to the permitted action. Similarly, access may be restricted to users who are not subject to ITAR restrictions, and the predicate in the Conditions field would evaluate the appropriate attribute in the user's record.

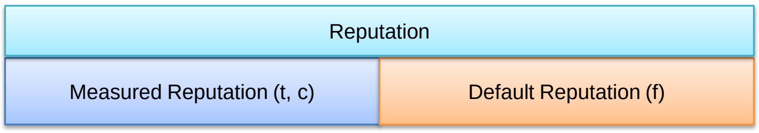


Figure 2: The Representation of Reputation in TrustForge

An example of a policy specification that takes reputation into account is shown in Figure 1. It can be seen that KeyNote is expressive enough satisfy our design goal of being able to specify an application-specific policy. For example, we use the Conditions field to include reputation values into policies. Here, predicates in the Conditions field allows us to specify constraints (typically, lower bounds) on the reputation values that are necessary for access. The figure shows a policy that allows any user (denoted by the * symbol in the users field), who has reputation above 0.75, to check in new components, check out components, and delete them, but not edit existing components. Almost any other attribute-value pair can be added to the policy language in a similar manner.

2.2 Reputation Function

In order to keep the policies specified in KeyNote evolve, we use the notion of user reputation computed based on their contributions to the projects in the repository. Therefore reputation mechanism in TrustForge is basically identifies high-quality components and high-quality contributors. The design of reputation system is takes into account the presence of malicious users who may want to bias the reputation mechanism (*i.e.*, increasing their own reputation or decreasing the reputation of other users) by either introducing erroneous feedback or taking advantages of the loop hole of the reputation algorithm design. In this section, we will discuss our design of an effective reputation mechanism, which is also robust to potential attacks.

2.2.1 Reputation Representation

In TrustForge, the reputation of components and users is denoted as a three-dimension vector (t, c, f) by adopting the CertainLogic representation [7]. CertainLogic provides a novel model for the evaluation of the trustworthiness of complex systems under uncertainty, which is also compliant with the standard probabilistic approach. As shown in Figure 2, the first two dimensions (t, c) is called measured reputation (t is the measured value, c is the confidence value), which encodes the reputation value computed based on direct feedback. Meanwhile, the third dimension (f) is called default reputation, which encodes the reputation value computed based on indirect evidence or inference. Furthermore, two operators are defined for the reputation vector following the semantic of CertainLogic:

1. Fusion operator, which merges several reputation vectors into one. The detailed definition of the fusion operator is shown in Figure 3:

$$
t_B = \begin{cases} \dfrac{\sum_{i=1}^{n} t_{A_i}}{n} & \text{if } c_{A_1} = c_{A_2} = \cdots = c_{A_n} = 1 \ , \\ 0.5 & \text{if } c_{A_1} = c_{A_2} = \cdots = c_{A_n} = 0 \ , \\ \dfrac{\sum_{i=1}^{n} (c_{A_i} t_{A_i} \prod_{j=1,\ j \neq i}^{n} (1 - c_{A_j}))}{\sum_{i=1}^{n} (c_{A_i} \prod_{j=1,\ j \neq i}^{n} (1 - c_{A_j}))} & \text{if } \{c_{A_i}, c_{A_j}\} \neq 1 \ . \end{cases}
$$

$$
c_B = \begin{cases} 1 & \text{if } c_{A_1} = c_{A_2} = \cdots = c_{A_n} = 1 \ , \\ \dfrac{\sum_{i=1}^{n} (c_{A_i} \prod_{j=1,\ j \neq i}^{n} (1 - c_{A_j}))}{\sum_{i=1}^{n} (\prod_{j=1,\ j \neq i}^{n} (1 - c_{A_j}))} & \text{if } \{c_{A_i}, c_{A_j}\} \neq 1 \ . \end{cases}
$$

$$
f_B = \frac{\sum_{i=1}^{n} f_{A_i}}{n}
$$

Figure 3: CertainLogic Fusion Operator

$$Fusion(A_1, A_2, \ldots, A_n) = (t_B, c_B, f_B)$$

2. Expectation operator, which computes a scalar reputation value based on the vector representation:

$$Expectation(A) = t_A * c_A + (1 - c_A) * f_A$$

2.2.2 Design of the Reputation Function

With the understanding of the reputation representation, we now describe the reputation algorithm used by TrustForge. We started by identifying all the feedback information source available, which can be used to evaluate the trustworthiness of user and the quality of component. For each feedback information source, we further identity a suitable algorithm, which can be used to infer the trustworthiness of components and users based on the corresponding feedback type. These algorithms serve as the basic building blocks. By properly combining them, we design a hierarchical algorithm that generates final reputation for both users and components as shown in Figure 4.

Feedback and Information Sources In total, we identified four independent feedback information sources for reputation computation, namely:

1. *Test Evidence:* The qualitative or quantitative test results on how well components satisfy their corresponding requirements.

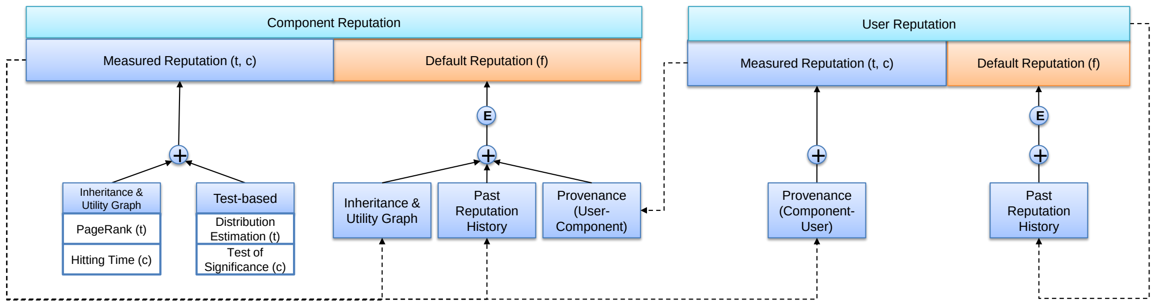


Figure 4: TrustForge Reputation Algorithm Overview

2. *Component Utility & Inheritance:* The information on how the components are reused as building block to compose more complex components, and how properties and designs of components are passed to other components.

3. *User-Component Provenance:* The information on components and their corresponding contributors.

4. *Revision History:* The information on how component and user reputation evolves over times.

All the feedback information is stored in the data model as discussed in previous section.

Component Reputation Calculation Figure 4 shows the skeleton and information flow of our reputation algorithm design, where nodes with the `+` and `E` symbol corresponds to the fusion and expectation operator, respectively. To compute the measured reputation of component, we have designed two basic building blocks:

1. *Inheritance & Utility Graph:* This building block uses the graph of component inheritance and utility relationships based on the assumption that if a component is highly used or inherited by other components in the system, then it is of high-quality. The t value is computed based on PageRank algorithm [6] over the graph of inheritance and utility links between components. The c value is assigned based on a graph hitting time algorithm [3], which can prevent malicious users from adding meaningless links in the graph to unfairly increase their reputation.

2. *Test-based:* This building block is based on test evidence. The idea is that by using qualitative or quantitative tests, one can properly measure how well a component satisfies its requirements. The t value is computed based on statistical distribution estimations of the test evidence to get the probability of requirement satisfaction by a component. The c value is computed based on the test of significance of the corresponding statistical estimator.

The value computed by these two building blocks will be merged using the fusion operator to get the measured component reputation. Further, in order to compute the default reputation of component, we have designed three building blocks:

1. *Inheritance & Utility Graph:* This building block uses the same component inheritance & utility graph, but focuses on the out-going links instead of in-coming ones as in the computation of measured component reputation. The idea here is that if a component inherits or reuses code/design from other high-quality component, then this component is more likely to be of high quality, and *vice versa.* To compute this building block, we will fetch the measured reputation of all the components from which one component uses or inherits code.

2. *Revision History:* This building block uses the revision history information of components. The assumption is that if a component is high-quality in the past revisions, it will still be high-quality in the future, and *vice versa.* To compute this dimension of reputation for a component, we will fetch its past measured reputation values.

3. *Provenance (User-Component):* This building block uses the provenance information between a component and its contributors. The idea is that high-quality components are contributed by trustworthy users, and *vice versa.* To compute this dimension of reputation for a component, we will fetch the measured reputation of all its contributors.

The reputation values fetched by these three building blocks will be merged using the fusion operator. The default reputation of component is the expectation value of the aggregated value.

User Reputation Calculation The measured user reputation is computed using the provenance information between a user and all the components he/she has contributed to. The assumption here is that high-quality components are contributed by high-quality users, and *vice versa.* To compute this dimension of reputation, we will fetch the measured reputation of all components a user has contributed to, and merge these values using the fusion operator.

To compute the default reputation of user, we use the revision history information. The assumption is that if a user contributed high-quality components in the past revisions, he will keep doing so in the future, and *vice versa.* To compute this dimension of user reputation, we will fetch historical measured reputation of a user, merge these values using the fusion operator, and take the expectation of the aggregated value.

Once the reputation values have been calculated, they are plugged into the KeyNote policy and the access control decisions are made for the individual users.

2.2.3 Attack model and defense strategies

Thus far, we have described on the basic reputation algorithm design used by TrustForge. By using quite extensive feedback information and properly combining them together, the reputation algorithm can effectively identify high-quality components and trustworthy users. Here, we will look at its robustness. The problem we are trying to address here is that malicious users may manipulate the reputation mechanism by introducing biased feedback information. Therefore, we need to identify potential attacks to the our reputation algorithm, and design proper defense mechanism as protection.

We first enumerate the attack models being considered in the TrustForge system. In summary, an attacker want to unfairly boost his own reputation or decrease the reputation of benign users, in order to grant more access privileges. Attacker can only achieve the goal by injecting biased information into the TrustForge system to mislead the reputation algorithm. Concretely, we have:

1. An attacker may introduce "dummy components" and "spamming links" into the utility and inheritance graph to unfairly increase the popularity of the components contributed by the attacker. So that the PageRank value of the attacker's components will increase, and the PageRank value of other's components will decrease.

2. An attacker may submit false evidence to show that his components satisfy the corresponding requirements, or to show that others' components fail to satisfy the corresponding requirements. As the result, the "test" dimension of the component reputation will change in favor to attacker's components.

As the countermeasure to the first type of attack, we compute a hitting time instead of the classic PageRank value as the countermeasure. As proved in [3], the introducing of outgoing spamming links won't increase the reputation of the attackers by using this defensive approach. To defense the second type of attack, we specify trust policies to only allow a set of predefined truthful users to submit test results by given them curate privilege.

Furthermore, we assume that for the CertainLogic operators adopted in TrustForge, if the operant values are truthful and the operation is performed by truthful entity (in this case, TrustForge), then the result is also truthful. Section 4 provides a more detailed analysis of TrustForge reputation function performance and demonstrates that it satisfies our design goals of being and robust.

2.3 TrustForge Data Model

In order to perform calculation of reputations, TrustForge needs to store certain metadata about the repository. This metadata is designed to capture three key aspects: the *provenance* of the objects in the repository, *i.e.,* who contributed them and who made modifications to them; the *usage* of the components, *e.g.,* as subcomponents of other components; and any *test results.*

TrustForge captures provenance information implicitly by observing checkins. Analogous to other version control systems, users can check in new objects, modify or delete existing ones, and create or merge branches. Based on these checkins, TrustForge builds a kind of "family tree" for each object in the repository that describes all the modifications that were made during the object's lifetime, as well as the corresponding time and the user who made them. A similar "family tree" at the component level captures usage and inheritance.

Since both provenance and usage are inherently graph-structured, the decision was made to store the data as a graph, rather than in a relational database. The graph contains a vertex for each user, component, and checkin, as well as for each of an object's revisions; the edges describe relationships between vertices – for instance, that a particular revision was created by a specific checkin, or that a given component is part of another component. Test results are not part of the graph; rather, they are represented as annotations to particular vertices.

2.3.1 Vertices and edges.

TrustForge's data model contains four types of vertices – `user`, `component`, `checkin`, and `object` – as well as five types of edges:

- A `derivedfrom` edge connects an object revision to other object revisions[1] from which it was derived;
- A `ispartof` edge connects an object revision to the checkin that created it;
- A `belongsto` edge connects a checkin to the component to which it was applied;
- A `performedby` edge connects a checkin to the user that performed it; and
- An `issubcompof` edge connects a subcomponent to its parent component.

The provenance-related vertices and edges are created automatically after each checkin. When user U checks in modifications to some objects $o_1, \ldots, o_k$ of a component C, TrustForge creates a new `checkin` vertex v_c and k `object` vertices $v_{o,k}$, one for each modified object; it then creates an `ispartof` edge from each $v_{o,k}$ to v_c, a `belongsto` edge from v_c to C, and a `performedby` edge from v_c to U. The vertices are then annotated with some metadata, e.g., a revision number. The usage edges and the annotations for test results are created manually, in response to user actions.

Note that the graph is append-only: existing vertices and edges are never modified or removed. This is a security feature: since information is never lost, dishonest users have no way to remove telltale information. Of course, this means that TrustForge's storage requirements will grow over time, but they will not grow faster than those of the underlying repository, which never deletes past revisions either. If necessary, old revisions can be removed manually by the operators.

2.3.2 Query language.

Although the primary function of the graph is to provide the input for the reputation function in Section 2.2, we expect that it will have other uses – e.g., to perform forensics when bad

[1]When branches are merged, the resulting objects are derived from their counterparts in each branch.

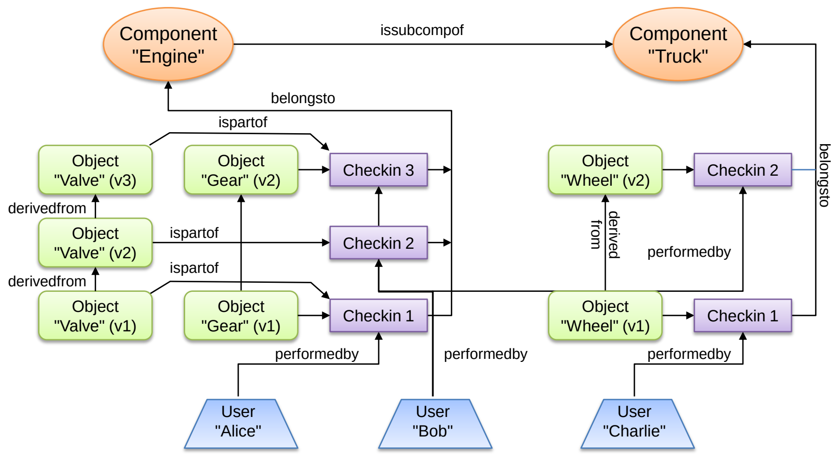


Figure 5: The TrustForge data model. This simple repository contains two components, a truck and its engine, which were contributed by Alice, Bob, and Charlie.

contributions are discovered, or to search for potential contributors for a new project based on the users' prior expertise. Furthermore, TrustForge's reputation function may evolve over time, e.g., to support new types of tests or new kinds of contributions. Therefore, instead a hard-coded interface to the graph, TrustForge contains a general-purpose query language that can be used to formulate a wide variety of queries.

TrustForge's query language is an extension of ProQL [5, 4], a query language that was specifically designed for provenance graphs. ProQL queries use a *path expression* syntax: each query describes a set of paths in the TrustForge graph, and then performs some computation on those paths. For instance, we might be interested in the "impact" a given user U has made on the repository so far. We might then describe the paths that start at U's vertex, traverse the `performedby` and `belongsto` edges to reach the an object revision U has contributed, then traverse potentially multiple `belongsto` and `issubcompof` edges, and finally terminate at some component C that (perhaps transitively) uses that revision. We might then weight each path based on the magnitude of the revision and the reputation of the component C, and finally aggregate the contributions from the different paths.

The original ProQL language did not support iterative computations similar to PageRank, which is an important building block in TrustForge's reputation function. Therefore, we extended ProQL with a `repeat...until` construct (which can express the iteration and its termination condition) as well as with support for propagation and adjustment of vertex annotations (which can carry the page ranks). With these extensions, we can express the entire PageRank computation in a short ProQL query, thus meeting our design goal of having a efficient data engine. A more detailed description of these extensions is available in [4].

2.4 Interface to Component Repository

We have implemented TrustForge as an independent stand-alone entity that interacts with the component repository through a set of standardized APIs. The purpose of the APIs is two fold: (1) to collect information from the `VehicleFORGE` repository about the users within the system and the components they contribute; and (2) to expose the capabilities of TrustForge for the repository to use, such as specifying access policies and making access control decisions. Note that, TrustForge by itself is not the system-of-record for component and user attributes (except reputation). It only stores a copy of the information. Further, TrustForge is not an outward facing system. It therefore does not implement any security features for the APIs as the assumption is that all the access to TrustForge APIs will be filtered through the `VehicleFORGE` repository. All data exchanged between TrustForge and `VehicleFORGE` is in textual format. We use the JSON format[2] for data exchange due to its lightweight nature and ease of use. We create a simple REST API for accessing TrustForge's capabilities for `VehicleFORGE`. For each of the API calls, we specify the JSON schema of the messages exchanged, and the specific URL to which the call needs to be made.

In general, the interaction between TrustForge and `VehicleFORGE` repository through the API can be classified into two categories: information insertion and information extraction. Information insertion into the TrustForge takes the form of: (1) **INS1**: updates regarding changes in the repository elements such as projects, components and their interconnectivity in terms of mutual usage, (2) **INS2**: updates regarding the results of tests conducted on the components as evidence of satisfying the underlying requirements, (3) **INS3**: updates regarding new users who have joined `VehicleFORGE`, including attributes such as identification, citizenship, location and so on, and (4) **INS4**: updates regarding access control policies active within a project. Similarly, information extraction from the TrustForge is in the form of: (1) **EXT1**: access requests asking for permission to perform a specific task by a user (returns yes/no response), (2) **EXT2**: queries related to the policies associated with the various projects in `VehicleFORGE`, and (3) **EXT3**: queries the reputation values of entities involved.

The TrustForge API document gives more details about the JSON schema that goes with each of the API calls. Each INS API call returns a 201 HTTP message for each of the INS APIs. It returns a 401 code if the task is unauthorized and 404 if the URL is not found. The TrustForge system also returns specific warnings back with the 201 message in case something untoward is observed, for example circulation delegations. For the EXT API calls, a 200 HTTP message to demonstrate successful query. Other codes remain the same. The following is the list all the APIs that are provided by TrustForge and the categories they fall into.

- `TrustForge_update-feedback(update)` [INS1/INS2] : The TrustForge expects updates about components to be pushed to it. This API primarily deals with populating the data model for reputation computation. Updates can be of three types: specifying component changes, specifying component test result, and specifying component use

[2] `http://tools.ieft.org/html/draft-zyp-json-schema-02`

relationships.

- `TrustForge_update-user-default(user_id,[attrib_name,attrib_value])` [INS3]: This API is used to provide information to TrustForge about a new user including her $user_id$, and a name value pair of attributes such as $[citizen, US]$ and $[itar, yes]$.

- `TrustForge_policy-update(user_id,lic_id,proj_id,cred_name,conditions,prior)` [INS4]: This API allows a user to create and update the policy associated with a project, *project_id*. Policies physically manifest themselves as delegations (from project manager to users and then further on). There can be multiple such delegations from project managers and other users, associated with a project. Each delegation results in the creation of a KeyNote credential with a unique ID within the project. Here, the *user_id* specifies the authorizer, the *lic_id* specifies the licensee, *cred_name* specifies the name of the delegation, and *conditions* specify the context in which the delegation acts.

- `TrustForge_access-request(areqid,user_id,comp_id,proj_id,cred_list,access_type)` [EXT1]: This API checks to see if a user with *user_id* can obtain access of *access_type* (credential < *create*, *read*, *write*, *delete*, *curate* >) on a component, *comp_id*, within a project, *proj_id*, and returns a yes/no answer. The attribute *cred_list* is used to determine which delegation(s) associated with the project policy is (are) used in the access request. The attribute *areqid* is used to uniquely identify the access request. It is the responsibility of repository to match the response returned to the request sent. Note that, search does not require a separate privilege.

- `TrustForge_policy-query(pqueryid,proj_id,qtype,user_id)` [EXT2]: This API determines the policy (list of all credentials) associated with the project, *proj_id* provided the user, *user_id* belongs to the authorizer or licensee (specified by the *qtype* element). The credential could be a result of issuing project wide policies as a program manager. They could also be generated a result of a user delegating components within a project to others. The attribute *pqueryid* is used to uniquely identify the policy query request. Returns a list of credentials.

- `TrustForge_delegation-query(dqueryid,cred_name)` [EXT2]: The API returns the details of a delegation, given its unique id as specified by the variable *cred_name* (delegation credential name). This is usually used to modify the policy of a project.

- `TrustForge_log-query(plogqueryid,user_id,proj_id, query)` [EXT 2]: The API allows the querying of the logs (policy log, component modification log, access log) maintained for project identified by *proj_id*. The query is authorized only if the user with *user_id* has project manager privileges. The response to the query is expected to be an array of attribute-value pairs. The attribute *plogqueryid* is used to uniquely identify the log query request.

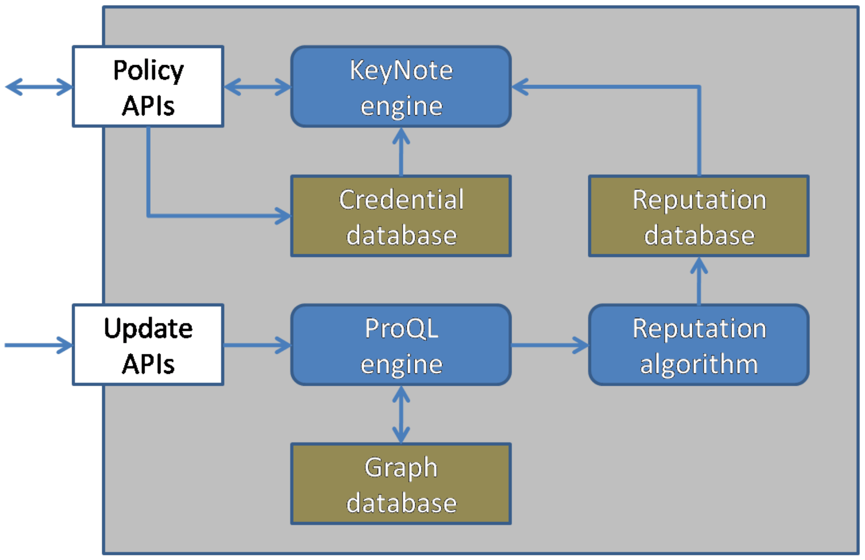


Figure 6: Architecture of the TrustForge implementation

- `TrustForge_query-userrep(user_id)` [EXT3]: This API allows the querying of the reputations of a user, given by *user_id*.
- `TrustForge_query-comprep(comp_id)` [EXT3]: This API allows the querying of the reputations of a component, given by *comp_id*.

3 TrustForge Implementation

We have implemented the design outlined in the preceding sections as a web service. The service is running on Linux and deployed using Amazon EC2 virtual machines. The use of virtual machines allows us to easily deploy multiple instances of TrustForge as needed.

The architecture of the TrustForge implementation is shown in Figure 6. The web interface is implemented in python and is running within an Apache web server. Policy updates and queries are handled by an off-the-shelf implementation of the KeyNote engine. Policies, as well as user attributes and reputation values that are used by KeyNote in policy evaluation, are stored in a MySQL database. Relationships between components in the repository are stored as a graph in a separate graph database. The graph is queried using a ProQL query language. The ProQL engine is implemented in C++ over the BerkeleyDB storage system that keeps the graph database. Finally, the reputation algorithm, implemented in Java, uses ProQL queries to extract the relevant information from the graph database, performs calculation of user and component reputations and updates the reputation database.

4 TrustForge Evaluation

Now that we have described TrustForge, in this section, we will evaluate its capabilities in terms of differentiating the good users and components from the bad ones. In this regard, we first present a simulation based test-harness for TrustForge and then present the performance analysis results obtained.

4.1 System-in-the-Loop Simulator

In order to test the efficacy of the TrustForge policy engine and the reputation computation and fine tune its performance we built a simplified `VehicleFORGE` simulator. The purpose of the simulator is to mimic the basic capabilities of the `VehicleFORGE` repository in terms of providing the TrustForge with updates on the users and their components, specifying policies and performing access control. This *system-in-the-loop* approach allows us to evaluate and tweak our system to achieve the goal of increasing the contribution of good components while decreasing poor ones within the `VehicleFORGE` repository. Consequently, the principal capabilities of the simulator that we are therefore looking for are: (1) ability to exercise the APIs provided by TrustForge, (2) ability to provide updates regarding components and their inter-connections, (3) ability to add tests results to components, and (4) ability to simulate various user types with varying degrees of maliciousness.

Conceptually, one can imagine the simulator system to be trying to enable introduction of components and assemblies of components within a component exchange or repository. Assume there are three users within the system $U1$, $U2$, $U3$. Each user has one or more (atomic or composite) components that it has contributed to the system. Assume $U1$ has contributed $1 : X$, $1 : Y$ and $1 : Z$, $U2$ has contributed $2 : V$ and $2 : A$, and $U3$ has contributed $3 : X$, all of which are atomic. Note that, $1 : X$ and $3 : X$ are components built to the same specification, just by two different users. Now assume U2 wants to introduce a new composite component $2 : P$ into the system. Now $2 : P$ requires the use of component X already in the system. In order to do that, it first reads one of the two X components (namely, $1 : X$ and $3 : X$) available with in the repository, decides which one meets its requirements, uses it to construct $2 : P$ and then finally introduces $2 : P$ into the repository. The purpose of the simulator, in the simulator-TrustForge interaction, is to execute this very process using dummy components, while the task of the TrustForge is to accept of reject them pursuant to the policies currently active for the project within it.

4.1.1 System description

The Figure 7 illustrates the simulator system and its workflow, which moves from left to right. The simulator is designed on the principal of using *traces*, where a trace specifics up-front all the actions with respect to users and their components. The trace can be thought of as a pre-defined scenario with specific characteristics that is executed on TrustForge. This allows us to test the various functionalities of TrustForge and tweak it, while keeping the underlying context of operation static. The first step in the generation of a trace is the Type

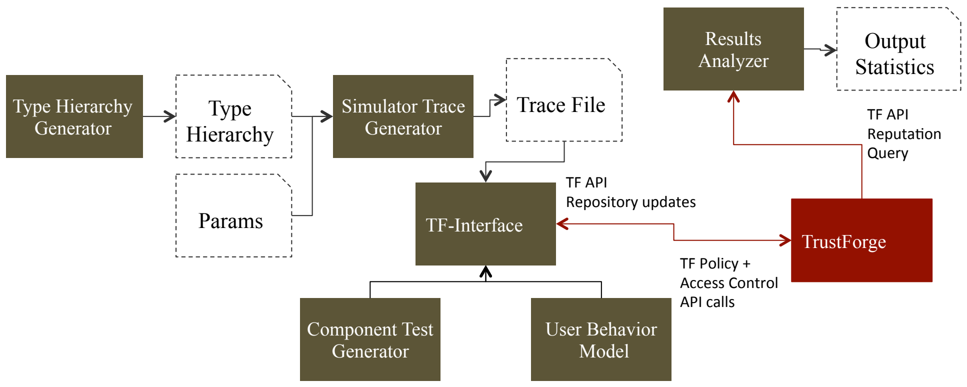
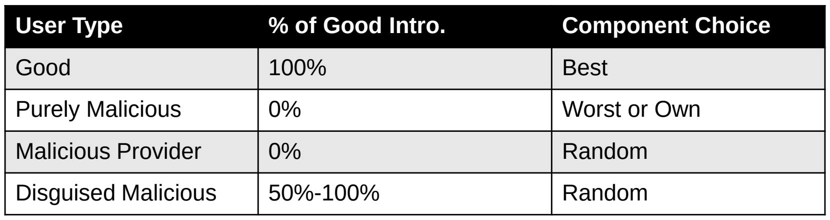


Figure 7: TrustForge Simulator System

Hierarchy Generator (THG) on the very left is where the figure. The THG takes as input the number of nodes and edges and generates an acyclic graph, known as the *type hierarchy*, that provides list of all the components, their different versions that will be used in the execution, along with the underlying components for component assemblies. Therefore an entry of the form"0[3]: 1, 2" indicates that the version three of component zero is a component assembly and will require component 1 and component 2. At least one instances of both component 1 and 2 will have to be available within the repository for TrustForge to permit a user to be able to successfully add component 0[3] to the repository.

4.1.2 Trace Files

The type-hierarchy is then used to generate a trace-file that lists a series of actions to be taken during that execution of the simulator, using the Trace Generator (TG) component. The TrustForge takes a number of parameters including — the number of users within the system, the types of users, and the number of transactions to be executed within this run. It then generates a list of component addition actions to the repository. There are as many entries as the number of transactions specified as input for the simulator. Entries in the trace-file are of the form $(U, C[v])$, where U is the user and C is the component name and v is the version. Therefore an entry of the form $(X, 0[3])$ mean that $userX$ wants to add a $component0$ at $version2$. Another type of entry in the trace-file is the test result for the components already in the repository. The simulator allows any user to contribute test results to the components in the system. The Component Test Generator (CTG) is responsible for adding the test entries in the trace-file. The entries for this are prefaced by the string 'T!' in the trace-file. The test results are provided in the (t,c) format used by CertainLogic as described in Section 2.3. We assume that all tests are done at 95% confidence level which forms the 'c' value, while the 't' value is randomly generated and approximates the ground-truth value of the component. The ground-truth is of course known to the simulator. Every component added to the system has a ground-truth value between 0 and 1. This value



User Type	% of Good Intro.	Component Choice
Good	100%	Best
Purely Malicious	0%	Worst or Own
Malicious Provider	0%	Random
Disguised Malicious	50%-100%	Random

Figure 8: TrustForge Simulator User Behavior Model

indicates their trustworthiness. Good components have values close to 1 and bad ones have values close to 0. If a good user is providing test results, then 't' value associated with the test will closely approximate the actual ground truth, while in the case of designated bad users providing test results, the 't' value may vary.

4.1.3 User Behavior Model

This brings us to the user model that is used by the simulator. From the previous discussion it can be seen that not all users adding components and tests with the system are good. By good we mean contribute high quality (reputation value close to 1) atomic components, use high quality base components for component assemblies. It is possible that the `VehicleFORGE` system has some potentially malicious users who try to add poor quality (low ground-truth value) components to the system and try to maintain high reputations values for themselves and their components. The idea behind the use of TrustForge is to be able to clearly distinguish such users and their components from the good users, who follow the rules. In this regard, we have designed a user model that captures various types of bad users. When simulator trace-file is generated, a pre-specified number of bad users are generated by the simulator. The table below shows the various types of bad users. Each user in this model has two properties with respect to adding atomic components (% of good components introduced) and choosing base components for adding component assemblies (chosen base). The good user adds good quality components 100% of the time, while choosing good base components. A purely malicious user does the exact opposite. A disguised malicious user only adds good components between 50%-100% of the time to disguise its maliciousness. We use the normal distribution to determine the probability of adding good components in a particular transaction for the disguised malicious users. Figure 8 illustrates the user model used by the simulator.

4.1.4 Execution

When executing an entrain the trace-file, the simulator calls the `Tf_access-request` API to first check to see if the user can 'create' a component to the repository. If possible, then it makes more access queries to the to see if instances of base components are present with the system and if so, does the user have 'read' access to them. The simulator then chooses a version each of the base components that is most conducive to the user, based on the underlying behavior model. For example, if the user is purely bad then it will try to choose its own or other poor quality base components. This improves the utility of bad component being used as the base, thus increasing their reputation and that of their authors. The simulator then sends an update to the TrustForge (through the `Tf_update-feedback` API) regarding the addition of the new component and its links to chosen instances of the base components to be stored in the ProQL database. When executing the test entries for components, the simulator first calls to `Tf_access-request` API in TrustForge to first check to see if the user can 'curate' a component to the repository. Test results are only accepted if the 'curate' privileges are available to the user. As the various transactions are being executed, the simulator using the Result Analyzer, from time to time, queries the reputation value of the components and users from TrustForge using the `TrustForge_query-userrep` and `TrustForge_query-comprep` API calls. These values are recorded and made available for further analysis.

4.2 Evaluation results

To exercise the simulator and the TrustForge infrastructure discussed in previous sections, we conduct experiments to evaluate the effectiveness of combining the policy and reputation perspectives for making access control decision. Particularly, we focus on evaluating the effectiveness of the reputation algorithm and its satisfaction of our design goal of having a robust reputation function.

4.2.1 Experiment Setup

To setup the experiments, we use the simulator to generate a component type hierarchy with 50 component-type nodes and 100 "component type to component type" links. For each experiment, we ran it over at least 1000+ revision iterations. And for every 10 revision updates, we re-compute component and user reputation based on the reputation algorithm discussed in Section 2.2. These configurations are fixed across the series of experiments we conducted. On the user side, we vary the percentage of different user types among user community (detailed user behavior models discussed in Section 4.1.3). For disguised malicious users, we set their probability of contributing good components to be 50% in most of the experiments, unless otherwise notified. At most 50% of good users are given curate privilege to submit test results, and we also vary the size of this trusted tester set in some of the experiments. Tests are generated by such trusted testers for components that are randomly picked from the repository. The measured trust value of test results is informed by the ground truth quality of the corresponding components with $\pm 10\%$ error. The confidence value of test results are uniformly set to be 95%.

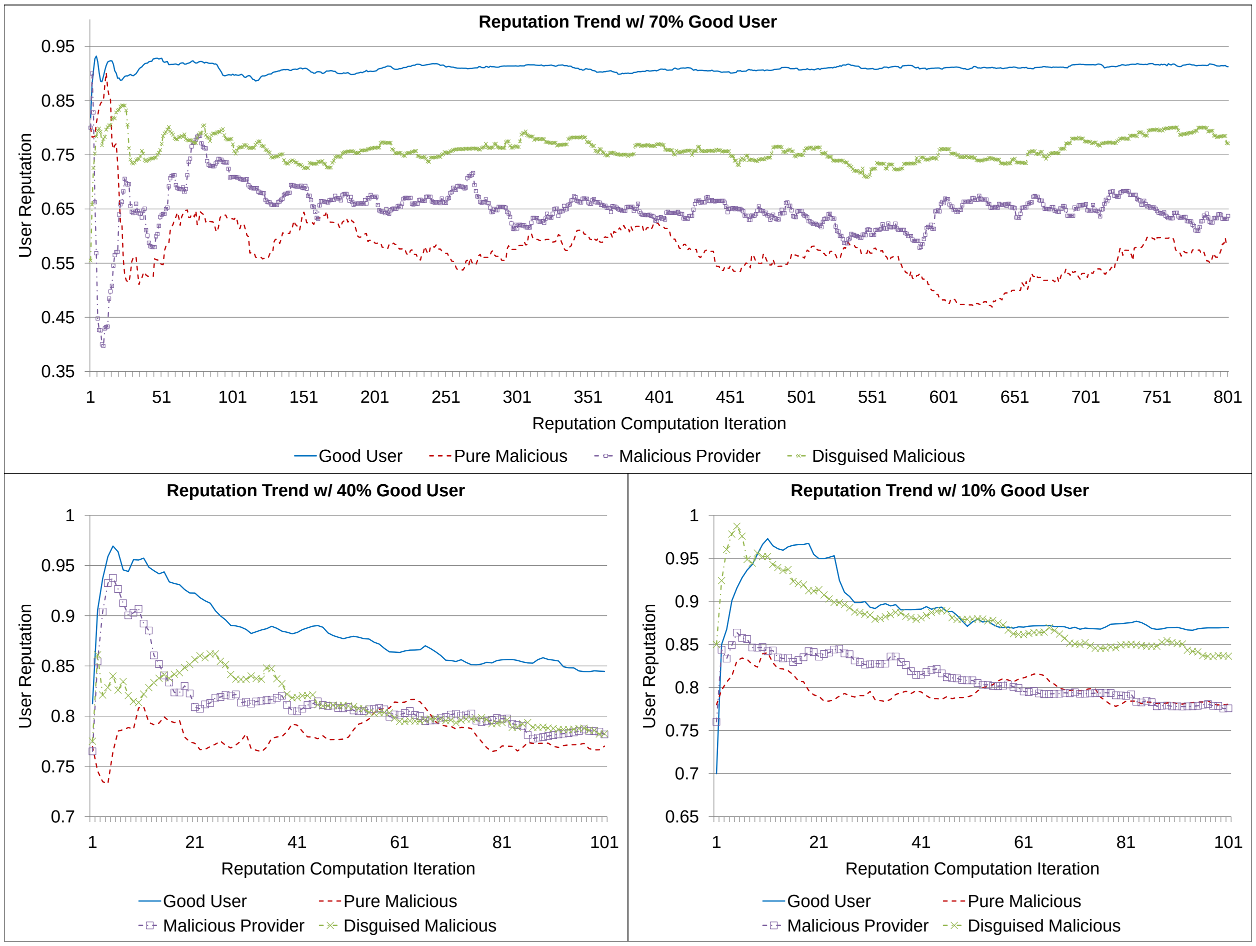


Figure 9: Overall Reputation Trends

4.2.2 General Reputation Trends

We first demonstrate the overall trend of user reputation computed by TrustForge. We vary the percentage of good users in the user community from 70%, 40% to 10%. Meanwhile, the corresponding percentage of total number of malicious users was increased 10%, 20%, and 30%. We run simulation experiments for each configurations over 8000 revision iterations (*i.e.,* 800 reputation computations). In Figure 9, we plot the average reputation trends for different user types over time. For clearer presentation purpose, we plot the full trend for the 70% good user configuration. For the other two configurations, we only plot the first 1000 revisions (*i.e.,* 100 reputation computations), which is good enough to capture the characteristics that we are interested in. Using the experiments, we observe that:

- *Convergence and Sensitivity:* The reputation curves quickly converge to a stable range for both good and malicious users after 20-30 reputation computation iterations. This shows that the sensitivity of the reputation function is good enough to capture user's behavior.

- *Effectiveness:* Further, we can observe a clear separation between different user behavior models. That is: (a) Good users often have much higher reputation than malicious ones; even when the good users are an absolute minority among the users. (b) Among the three malicious user types, the purely malicious users often get the worst reputation; both malicious providers and disguised malicious users get better reputation as they demonstrate more trustworthy behaviors than purely malicious ones. These results demonstrate that the reputation algorithm adopted in TrustForge is effective and can be used for making access control decisions.

4.2.3 Average Separation

With the overview of the reputation trends in mind, we further conducted experiments to investigate the reputation differences between good users and each of the three types of malicious users. In this regard, we created three user communities with good users and one type of malicious users – (good, purely malicious), (good, malicious provider), and (good, disguised malicious). For each user community, we vary the percentage of good users from 10%, 25%, 40%, 55%, 70%, 85%, 95%, and record the average reputation of good users and the corresponding type of malicious users from the 200th to 1000th revision iteration (*i.e.,* when the reputation value of user has converged to a rather stable range).

As shown in Figure 10, the reputation margin between good users and malicious users are wide enough for most of the scenario. Although the margin become narrow as the percentage of good user decreases within the community, we can still maintain meaningful margins even when this percentage is 30%. Further, although the malicious users can get very close reputation to the good users in some scenarios (especially, when malicious users is more than 85%), the average reputation of good user is still higher than malicious ones. These results provides very strong evidence that the reputation value computed by TrustForge can be used as a very dependable metrics for making access control decisions.

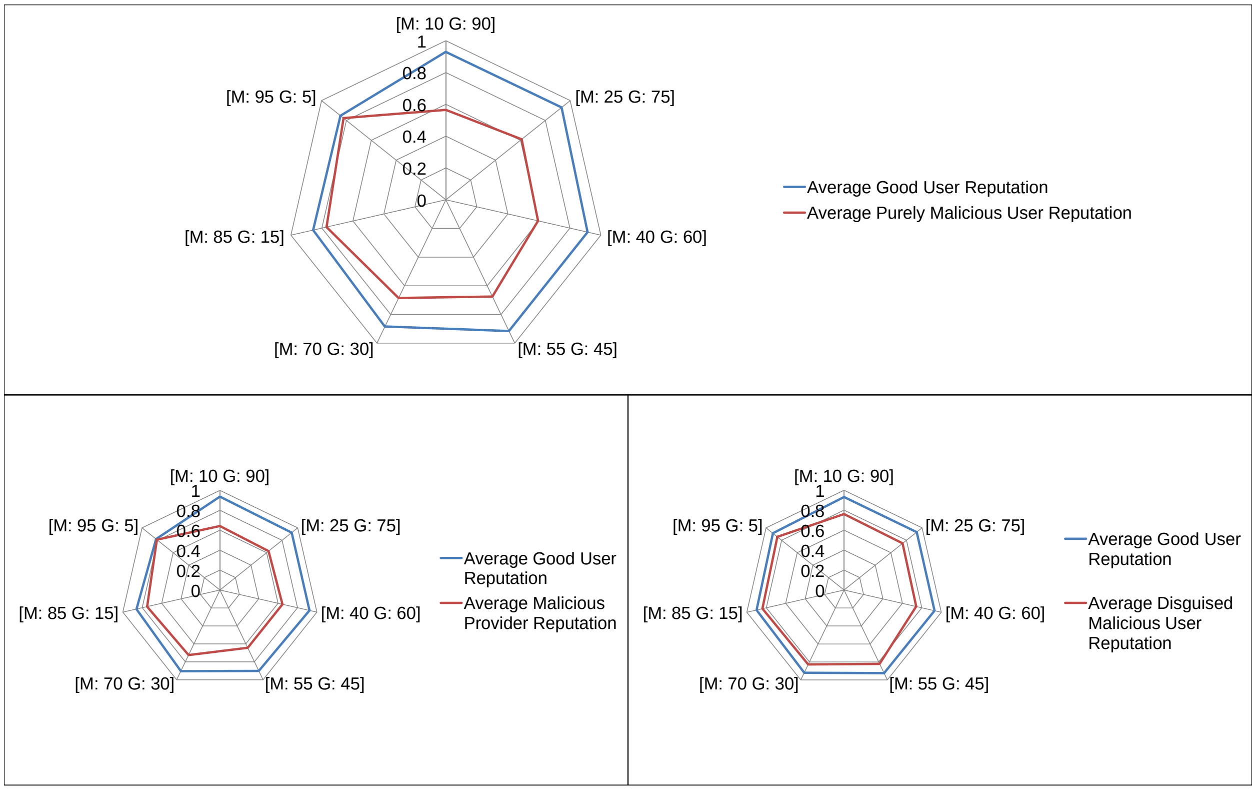


Figure 10: Average Reputation Margin Between Good Users and Malicious Users

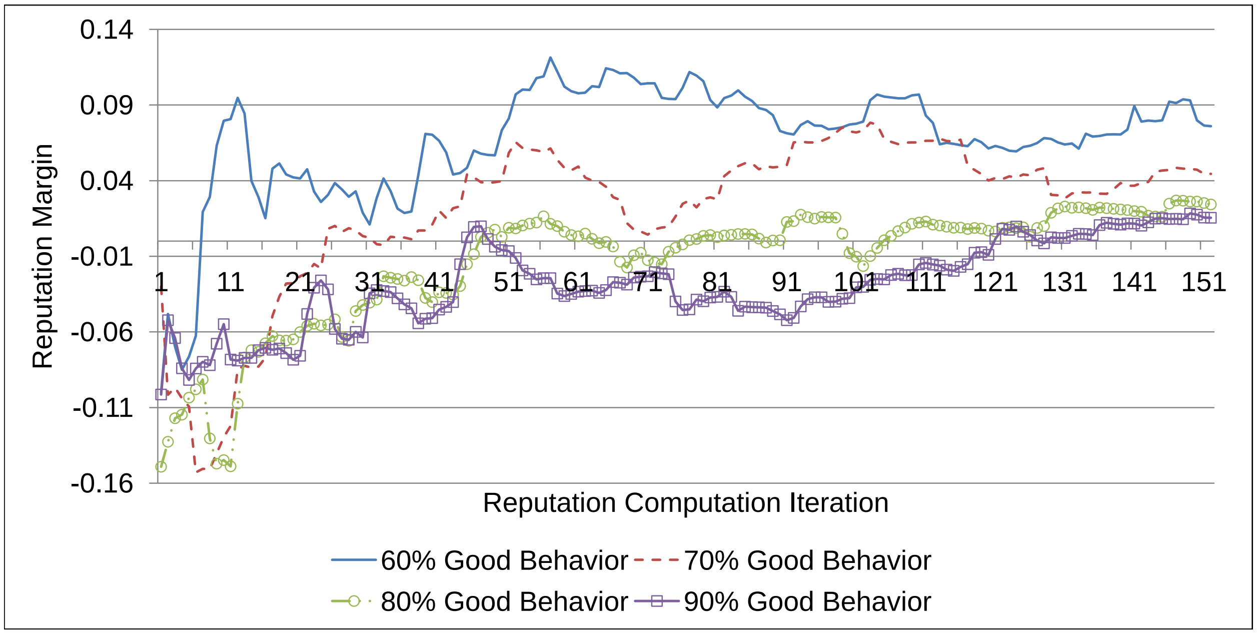


Figure 11: Minimal Reputation Margin Between Good Users and Disguised Malicious Users with Various Probability of Exhibiting Good Behavior

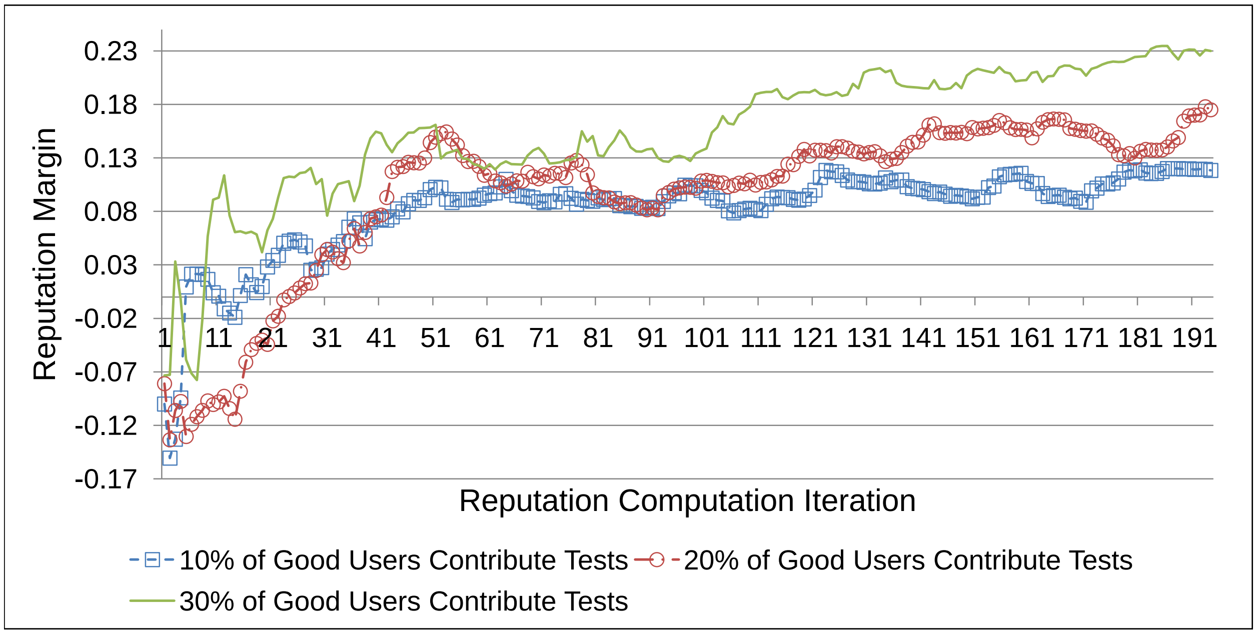


Figure 12: Minimal Reputation Margin Between Good Users and Purely Malicious Users with Various Amount of Tests

4.2.4 Minimal Separation

After a description of the average-case performance of the reputation mechanism, we switch our focus to the worst-case scenario. That is, we want to understand to what extent malicious users can game the system. As we see in previous experiments, the reputation margin between disguised malicious users and good users is the closest, since half of the time such malicious users demonstrate good behaviors. Therefore, we conduct further experiments by increasing the probability for disguised malicious users to exhibit good behavior from 50%, to 60%, 70% or even 90% (the percentage of good users in the community is set to be 70% in these experiments). Furthermore, we measure the margin between the disguised user who has the *highest* reputation value and the good user who has the *lowest* reputation value. In this way, we can clearly see the robustness of the reputation mechanism.

We illustrate the trends of this minimal reputation margin between good users and disguised malicious users in Figure 11. As we can see, as the disguised users exhibit more and more good behavior, essentially the observed behavior pattern of good users and disguised malicious users becomes very similar to each other. The reputation margin become smaller and it takes longer times and more informations for the reputation algorithm to discern the difference. Overall, we believe this result are satisfying, since the reputation margin is positive for most cases, and even when the disguised malicious users mainly exhibit good behavior.

4.2.5 Reputation under Limited Tests

We further conduct experiments to investigate the impact of the amount of available tests on the effectiveness of our reputation mechanism. In this experiments, we set up a user community with 70% of good users and 30% of purely malicious users. We then vary the size of the tester set, who are good users granted with curate privilege to submit test results, from 10%, 20%, to 30% of the total good users in the community (i.e., in contrast to 50% in our previous experiments). Furthermore, we measure the reputation margin between the purely malicious user who has the *highest* reputation value and the good user who has the *lowest* reputation value. By giving only limited test results to components, our reputation algorithm gets much less raw data that is used as the basis for the aggregate and computation of user reputation.

We illustrate the trends of this minimal reputation margin with limited test result in Figure 12. As we can see, after a few (about 5 - 20) initial iterations, the minimal separation between good user and malicious users is maintained. With less test results, the margin decreases, but is still wide enough for making correct access control decision. The results shown here provides a strong evidence that our reputation mechanism can work effectively even with very limited amount of test information. This result is very encouraging when one considers the application of TrustForge design to other open-source repository environments.

5 Conclusions and Future Work

In this report, we presented a access-control system called *TrustForge* that enables dynamic and flexible credentialing for collaborative-design component-based systems. In this regard, TrustForge automates the credentialing and access control process. It take a hybrid policy and reputation-based approach to address this problem. The policy language to specify the credentials for users in the system to contribute components. The reputation scores are then used to tune the credentials. Our implementation of TrustForge and its evaluation revealed its capabilities in terms of separating the good users and components from the bad ones. The next step in this regard would be to obtain real-life data from actual deployment of the system with the DARPA `VehicleFORGE` repository and fine-tune the reputation, data storage and policy engine further.

References

[1] Darpa adaptive vehicle make.

[2] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Conference of Security and Privacy*, Oakland, CA, 1996.

[3] John Hopcroft and Daniel Sheldon. Manipulation-resistant reputations using hitting time. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 68–81, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Zachary G. Ives, Andreas Haeberlen, Tao Feng, and Wolfgang Gatterbauer. Querying provenance for ranking and recommending. In *Proc. TaPP*, 2012.

[5] Grigoris Karvounarakis and Zachary G. Ives. Querying data provenance. In *Proc. SIGMOD*, 2010.

[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[7] Sebastian Ries, Sheikh Mahbub Habib, Max Mühlhäuser, and Vijay Varadharajan. Certainlogic: A logic for modeling trust and uncertainty (short paper). In *In Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*. Springer, Jun 2011.