ProProv: A Language and Graphical Tool for Specifying Data Provenance Policies

Kevin Dennis* Shamaria Engram† Tyler Kaczmarek† Jay Ligatti*

*Department of Computer Science and Engineering
University of South Florida
Tampa, USA
{kevindennis, ligatti}@usf.edu

† Secure Resilient Systems and Technology Group
MIT Lincoln Laboratory
Lexington, USA
{shamaria.engram, tyler.kaczmarek}@ll.mit.edu

***Abstract*—The Function-as-a-Service cloud computing paradigm has made large-scale application development convenient and efficient as developers no longer need to deploy or manage the necessary infrastructure themselves. However, as a consequence of this abstraction, developers lose insight into how their code is executed and data is processed. Cloud providers currently offer little to no assurance of the integrity of customer data. One approach to robust data integrity verification is the analysis of data provenance—logs that describe the causal history of data, applications, users, and non-person entities. This paper introduces ProProv, a new domain-specific language and graphical user interface for specifying policies over provenance metadata to automate provenance analyses.**

To evaluate the convenience and usability of the new ProProv interface, 61 individuals were recruited to construct provenance policies using both ProProv and the popular, general-purpose policy specification language Rego—used as a baseline for comparison. We found that, compared to Rego, the ProProv interface greatly increased the number of policies successfully constructed, improved the time taken to construct those policies, and reduced the failed-attempt rate. Participants successfully constructed 73% of the requested policies using ProProv, compared to 41% using Rego. To further evaluate the usability of the tools, participants were given a 10-question questionnaire measured using the System Usability Scale (SUS). The median SUS score for the graphical ProProv interface was above average and fell into the "excellent" category, compared to below average and "OK" for Rego. These results highlight the impacts that graphical domain-specific tools can have on the accuracy and speed of policy construction.

***Index Terms*—Data Provenance, Usability, Policy Specification**

I. Introduction

Cloud service providers have revolutionized computing by delivering applications and computing resources over the internet, allowing end users and application developers to accomplish tasks they otherwise may not have the resources or expertise to accomplish themselves. The Function-as-a-Service (FaaS) cloud computing paradigm has made the development of large-scale applications both convenient and efficient by allowing developers to build, run, and manage their applications as functions without having to deploy or manage the infrastructure required by such applications themselves. However, by outsourcing parts of the compute stack to third-party cloud providers, developers, as well as end users, of such applications lose insight into how their code is executed and data is processed—making it difficult to confirm the integrity of such data. Data integrity is a key metric for ensuring data correctness, quality, and reliability, which is becoming more and more important as the processing of data is outsourced. To facilitate widespread adoption of the FaaS computing paradigm for security-critical applications, additional tools are needed to give both cyber analysts and end users confidence in the integrity of their data.

Currently, end users of cloud services rely on data integrity checking techniques such as redundant data storage, RAID parity, and checksums [1], [2], and while these techniques can determine *whether* data was tampered with, they fail to provide more comprehensive details such as how and by whom. Such details are even more important for cyber analysts when filtering through security incident reports produced by threat detection software to distinguish between true incidents and false positives. For example, having insight into how a particular incident report was produced and what anomalous inputs triggered the report can be vital in responding to an incident in an appropriate and timely fashion.

Data provenance—logs describing the causal history of data, applications, users, and non-person entities—is a robust approach for the collection of information regarding the integrity of data. Such metadata is traditionally used as a forensic tool, due in part to the massive volume of log data that needs to be collected to gain a clear picture of cause-and-effect in a complex compute system.

The functional decomposition of monolithic applications into microservices that break down opaque workflows into a series of 'jobs' brokered by cloud service providers and executed in a hardware-agnostic manner is integral to the adoption of the FaaS paradigm for security-critical applications [3]. In the context of data provenance, this breakdown of function-

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

ality can be seen as atomic "activities", thereby facilitating lightweight provenance collection and reducing the volume of data to be analyzed to ensure data integrity. One such FaaS-like schema providing lightweight provenance collection is the Automatic Cryptographic Data Centric (ACDC) security paradigm [4].

By analyzing a provenance record, both developers and end users of ACDC applications can make stronger determinations about the integrity of their data. For example, an end-user of a batch-processing application may require that the application output is derived from a specific set of inputs. Engram et al. [4] present an approach called proactive provenance in which end users and developers can proactively reason about a datum's provenance record before consuming it for reading or computation. However, for such an approach to be effective, tools must be developed to automate such analyses.

This paper introduces ProProv, a new domain-specific language and graphical user interface for specifying policies over provenance metadata to automate provenance analyses. Such policies can be automatically evaluated by end users wishing to consume data returned from an application based on the data's provenance record. The ProProv language was designed with the goal of providing features not available in general-purpose policy specification tools to help users quickly and accurately develop provenance policies. Expert users can use the text-based language directly with conveniences such as syntactic sugar for common policy types and non-experts may use the graphical user interface to the ProProv language to assist with policy development. This paper also presents the results of a usability study that measured the effectiveness with which non-expert users could construct provenance policies using the ProProv interface.

The usability study was also conducted using Rego, a general-purpose policy specification language. The Rego language, described in more detail in Section IV-A, does not have a graphical user interface for defining policies similar to ProProv. To the author's knowledge, no existing industry-standard policy specification languages provide such an interface. GUIs limit the expressiveness of the tool, but are much more convenient to use, particularly for non-experts. The loss of expressiveness is acceptable, however, for a domain-specific tool like ProProv. The team hypothesizes that the convenience of the graphical interface will greatly improve efficiency and accuracy during policy specification.

This paper presents the following contributions: 1) ProProv, a language and graphical interface for constructing provenance policies, and 2) the results of a usability study comparing the ProProv graphical interface with Rego, illustrating the need for graphical domain-specific tools to improve the efficiency and accuracy of the policy specification process.

The rest of the paper is organized as follows: Section II provides background on data provenance and how it can be used to ensure data integrity, Section III introduces the ProProv policy language and interface used for the specification of provenance policies, Section IV provides the methodology for our study, Section V describes the results of the study, Section VI discusses the implications of the study results, and Section VII concludes.

II. Background and Related Work

This section discusses the importance of data integrity for ensuring the quality and reliability of data in computing systems and data provenance as a solution for comprehensive data integrity verification. We then examine prior work on provenance-based tools for various security-related tasks and show that little attention has been given to provenance-based tools for ensuring data quality *before* it is consumed for reading or computation. Lastly, we summarize related work on tools designed to simplify policy specification and discuss the lack of such tools for provenance policies.

A. Data Integrity and Data Provenance

Data integrity is a key metric for ensuring data quality and reliability—two characteristics important for ensuring the intended behavior of computer systems and their applications. Currently, the most popular mechanisms for ensuring data integrity include redundant data storage (i.e., mirroring), RAID parity, and checksums [1]. While these mechanisms can reveal whether data has been tampered with, they are inadequate for providing low-level details about such tampering, such as how and by whom.

To comprehensively assess data integrity, one must know the complete history of data, including all the manipulations it has underwent from its origin to its current state, how such manipulations occurred, and who influenced such manipulations. Two mechanisms for capturing such detailed data interactions include audit logs and data provenance.

The granularity at which audit logs capture data interactions vary from system to system, as they are manually configured by system administrators. System administrators typically configure audit logs to capture only the events that they deem security relevant. Because the set of security-relevant events considered by both a system administrator and system user may vastly differ, system users may, consequently, not be able to comprehensively verify the integrity of data they consume from the system. Furthermore, audit logs are generally not available to general users of computing systems and are typically used by system administrators to distinguish between normal and abnormal system behavior.

Data provenance is another way of recording the complete history of data, and a standard model for capturing such history is the W3C PROV Data Model [5], thus ensuring uniformity between systems. Data provenance is represented as a graph consisting of entities, activities, and agents. Entities represent data, activities represent functions that compute on data, and agents represent individuals or computer systems responsible for the state of data at a specific point in time [5]. The graphs also contain seven relations describing the causal relationships between entities, activities, and agents. A table describing the meaning of each relation and an illustration of the model can be found online [6].

B. Data Provenance Analyses

Several provenance-based analysis tools have been proposed for data-integrity checking. However, in most computing systems, data interactions can result in a mass of information that cannot easily be queried and analyzed in real time. As a result, many provenance-based analysis tools perform their analyses in retrospect, as a way of conducting a forensic analysis or security audit (e.g., [7]–[9]).

Provenance-based analysis tools can also be useful for a number of runtime-security checks [10], such as intrusion detection [11], [12], advanced persistent threat detection [13], [14], and data quality before it is consumed for reading or computation [4], [15].

FRAPpuccino [16] is a framework for detecting malicious behavior in cloud applications by building a benign-behavioral model using provenance data at runtime. FRAPpuccino can then detect malicious behavior when the execution of the application diverges from the benign model. NoDoze [17] is a tool for automatically triaging threat alerts based on runtime provenance analysis and is designed to be a more effective tool for threat detection by minimizing the number of false alerts for cyber analysts. While FRAPpucino and NoDoze are user-friendly, turn-key solutions for effectively detecting malicious runtime behavior, both tools rely on machine learning techniques and do not allow system administrators to specify application-specific policies to detect precisely defined behavior.

CamQuery [10] is a framework for analyzing live provenance at runtime to mediate security-relevant events; it allows users to construct policies using a vertex-centric API based on the graph-processing frameworks GraphChi [18] and GraphX [19]. While CamQuery is an effective framework for the construction of runtime provenance policies to mediate security-relevant events, it requires expert knowledge of sophisticated graph processing frameworks. We argue that in order for provenance-based analysis tools to be widely adopted, they must provide users with the flexibility to specify application-specific policies while remaining usable by non-experts and have a low-barrier to entry.

Proactive provenance is a runtime provenance-based method for verifying the integrity of data before it is consumed for computation in the Automatic Cryptographic Data-Centric Security (ACDC) architecture [4]. ACDC leverages various technologies to ensure the security and integrity of data in an environment similar to the FaaS paradigm.

ACDC coarsely captures provenance metadata in its environment, thereby limiting the explosion of provenance graphs. That is, ACDC provenance captures the executing function, a reference to the function's implementation (i.e., the defining contract), the function's inputs and output, a reference to the node agent performing the computation, and a reference to the account agent(s) responsible for a function's execution. ACDC provenance does not capture the internal execution steps of functions nor the internal workings of node agents. This is because functions are defined by contracts and therefore can be analyzed by static or dynamic code analysis tools; node agents are treated as black-box execution engines, but can also be subject to code analysis and tools purposed for the analysis of the node's hardware. While code and hardware analysis are important for ensuring a secure system, such analyses are outside the scope of, and should be done in conjunction with, provenance-based analysis. For a more detailed review of ACDC and its provenance model, we refer the reader to [4].

This paper extends the work presented in [4] by introducing a language and user interface for expressing proactive provenance policies. The user interface is designed to make it easier for non-expert policy writers to express provenance policies. While the user interface is designed specifically for provenance policies in ACDC, based on the results presented in this paper we believe the design principles of user-friendly interfaces for non-experts are broadly applicable to existing provenance-based analysis frameworks.

C. Tools for Specifying Complex Policies

Constructing security policies is a challenging and error-prone task, typically requiring policy writers to reason about all possible security-relevant actions, and in the context of provenance policies, all possible data interactions. Manually constructing these policies can be difficult due to the complexity of policy-specification languages and the sheer volume of security-relevant actions that need to be considered. This is a recognized problem in the literature, even for expert policy writers, and for non-expert policy writers the issue is exacerbated [20]–[22].

Several domain-specific policy management and visualization tools (e.g., access control and firewall configuration) have been proposed to ease the burden of constructing correct policies. Such tools vary widely on how they simplify the policy-specification process. For example, the Firewall Policy Advisor [23] analyzes large specifications and alerts the policy writer upon finding redundant policies or conflicts, it also provides a policy editor to assist policy writers when making modifications. Brostoff et al. [24] developed a policy-specification tool that incorporates education of access-control policies to increase policy writers' understanding of how to construct correct policies and found that the instructional features improved the efficiency with which expert policy writers could construct policies and the correctness of policies constructed by non-experts. Other tools use visualization techniques to increase understanding, efficiently spot errors, and ensure constructed policies align with the policy writer's intent [25]–[27].

PoliSeer [27] is the most closely, visually related tool to the ProProv interface. PoliSeer is a graphical user interface to the Polymer policy specification language [28]—a language for implementing application-specific runtime security policies for Java applications. PoliSeer relies on expert policy writers to upload policy specifications to the tool, after which non-expert policy writers can compose policies from the provided policy suite. When policy writers make a selection from the policy suite, the tool will parse the specification and display the policy to the user in a tree structure, which may include a tree with modifiable leaf nodes, to indicate to the user how to correctly construct the policy. Policy construction may include making an additional selection from the policy suite or typing in policy-relevant information. ProProv differs from PoliSeer in that 1) the ProProv interface is designed specifically for provenance policies and not application-specific policies for Java applications, and 2) ProProv does not rely on expert policy writers to provide pre-defined policies for use by non-expert policy writers.

To the best of our knowledge, almost no attention has been given to tools that simplify the policy-specification process for provenance policies. Due to the importance of ensuring data quality and reliability, it is important that provenance policies be specified correctly and according to the policy writer's intent. ProProv is designed to address these issues.

III. The ProProv Language and Interface

ProProv consists of two components: 1) a domain-specific policy specification language, and 2) a graphical user interface for constructing policies in the language. This section introduces the ProProv language and interface.

A. ProProv Language

The ProProv language is a first-order query language, where queries are assertions on provenance graphs. These queries define policies that specify sets, or instances, of provenance graphs that satisfy the expected provenance of data before it is to be consumed. When policies are evaluated over a specific provenance graph, they evaluate to either $true$ or $false$. A policy evaluating to $true$ indicates that the provenance graph under consideration satisfies the policy. Conversely, a policy evaluating to $false$ indicates that the provenance graph under consideration violates the policy.

Figure 1 shows the core syntax of the ProProv language. The language contains types for each ACDC provenance node type, provenance node lists, and booleans. The edge operators consist of all possible graph edge labels and can be constructed as a policy by following the edge operator with two provenance nodes enclosed in parentheses and separated by a comma. For example,

$$wasDerivedFrom(AverageSalary, EmployeeSalaries)$$

represents a policy that ensures a *dataEntity* named *AverageSalary* was derived from a *dataEntity* named *EmployeeSalaries*, which represents a data set containing all employee salaries. Such a policy is useful for ensuring that function outputs are derived from expected function inputs. Additionally, other possible policies in the language include boolean policies, negation policies, conjunction policies, universal policies, and universal policies with lists.

A boolean policy is specified with either the value $true$ or $false$. A $true$ policy is an "allow-all" policy. Such a policy means to trust, or consume, all data because $true$ means that the provenance graph under consideration satisfies the policy; therefore, a $true$ policy explicitly states that all graphs satisfy the policy. Conversely, $false$ is a "disallow-all" policy, which means to reject all data. Negation policies can be constructed by prefixing a policy with the ! operator. This policy evaluates to $true$ iff the policy following the ! operator evaluates to $false$. Conjunction policies combine two policies via the $\wedge$ operator such that both policies must evaluate to $true$ when evaluated over a provenance graph. For example, a policy writer might consider $AverageSalary$ to be a trusted $dataEntity$ iff it was derived from expected input $EmployeeSalaries$ and was attributed to the company's human resources department HR. Such a policy can be specified as a conjunction policy in the following way:

$$wasDerivedFrom(AverageSalary, EmployeeSalaries) \\ \wedge\, wasAttributedTo(AverageSalary, HR).$$

Universal policies assert that a policy p is true for all provenance nodes of a specified type. For example,

$$forall\ x : dataEntity.wasAttributedTo(x, HR)$$

ensures that every $dataEntity$ in a given provenance graph was attributed to HR. Lastly, universal policies with lists allow policy writers to assert that a policy p is true for all provenance nodes in a given list. A provenance node list can be constructed by using the :: operator to prepend nodes to the list. All provenance nodes in the list must have the same type. For example, consider the policy

$$forall\ x : dataEntity\ in\ AverageSalary \\ :: EmployeeSalaries :: nil.wasAttributedTo(x, HR).$$

This policy ensures that $AverageSalary$ and $EmployeeSalaries$, which have type $dataEntity$, were attributed to HR—nil is the empty list. Universal policies with lists may also be useful when policy writers want to constrain applications from consuming data with known untrustworthy interactions. Such a policy can be constructed by including all untrustworthy account agents, for example, in a list.

While Figure 1 shows the core syntax of the language, some policy writers may need to existentially quantify over variables or express disjunction or implication policies. For convenience, the language contains syntactic sugar for these types of policies. Existential policies can be expressed using the following syntax: $exists\ x : \nu.\ p$ (existential policies with lists can be constructed similarly). Disjunction policies can be expressed by including the $\vee$ operator between two policies (i.e., $p_1 \vee\ p_2$) and mean either p_1 or p_2 should evaluate to $true$. Implication policies can be expressed by including the $\Rightarrow$ operator between two policies (i.e., $p_1 \Rightarrow\ p_2$) and mean p_1 implies p_2. However, such policies can still be constructed using the core syntax. Existential policies can be constructed by combining the negation and universal policies. Disjunction and implication policies can be constructed by combining the negation and conjunction policies.

To specify provenance policies correctly, in a way that aligns with the policy writer's intent, the policy specification process requires policy writers to think about all the ways in

$$
\begin{array}{l}
\text{Nodes } n ::= \mathcal{E}_d \mid \mathcal{E}_c \mid \mathcal{E}_k \mid G_a \mid G_n \mid A \\
\text{Node Types } \nu ::= \mathit{dataEntity} \mid \mathit{contractEntity} \mid \\
\qquad \mathit{keyEntity} \mid \mathit{accountAgent} \mid \\
\qquad \mathit{nodeAgent} \mid \mathit{activity} \\
\text{Node Lists } \ell ::= \mathit{nil} \mid n :: \ell \\
\text{Booleans } b ::= \mathit{true} \mid \mathit{false} \\
\text{Types } \tau ::= \mathit{bool} \mid \nu \mid \nu_{list} \\
\text{Edge Operators } \oplus ::= \mathit{wasDerivedFrom} \mid \\
\qquad \mathit{wasAttributedTo} \mid \\
\qquad \mathit{wasGeneratedBy} \mid \mathit{used} \mid \\
\qquad \mathit{actedOnBehalfOf} \mid \\
\qquad \mathit{wasAssociatedWith} \mid \\
\qquad \mathit{wasInformedBy} \\
\text{Policies } p ::= b \mid !p \mid p_1 \wedge p_2 \mid \mathit{forall}\ x : \nu . p \mid \\
\qquad \mathit{forall}\ x : \nu\ \mathit{in}\ \ell . p \mid \oplus(n_1, n_2)
\end{array}
$$

Fig. 1. ProProv language core syntax

which relevant data may be considered (un)reliable, whether it be identifying all the agents who may have influenced the current state of data or identifying all the ways in which data may be processed or manipulated—manually doing so can be difficult and error prone. Additionally, policies are typically large and complex with sub-components potentially interacting in unexpected ways, which may result in policy conflicts or definitions that may not align with the policy writer's intent. For advanced policy writers using the text-based language, we use the Z3 SMT solver [29] to help detect when policy specifications contain conflicts. For example, a policy writer may inadvertently construct a policy that both allow and disallow certain data interactions (i.e., $p \wedge !p$). Such a mistake is common, even among expert policy writers [30]. For non-experts, we aim to simplify the policy specification process with a graphical user interface to the ProProv language.

B. ProProv Interface

The ProProv interface is a standalone front-end graphical user interface to the ProProv language; it is designed to make the language more accessible to non-expert policy writers. Because the intended audience for the interface is non-expert policy writers, it abstracts from the complete text-based language and provides users with the most basic features to construct meaningful policies and prevent the construction of policy specifications that may not align with their intent. Specifically, the interface does not include boolean policies (i.e. the "true" or "false' policy), as this tool is intended for non-expert users, and such policies represent an out-sized risk for misconfiguration. For example, a true policy means allow all data, essentially disregarding a datum's provenance record, and can result in consuming poor-quality and unreliable data. Conversely, a false policy means disallow all data and can potentially render a system inoperable because it will reject all data. Secondly, quantified policies with lists are also considered an advanced feature because all of the nodes contained in the list must have the same type, specifically the type of the variable that is quantified over. If a user includes, for example, a node of type $\mathit{keyEntity}$ in a list where all nodes must have type $\mathit{dataEntity}$, the language will treat the $\mathit{keyEntity}$ as a $\mathit{dataEntity}$ and not a $\mathit{keyEntity}$ as the user intended. This is because the variable quantified over in the policy has type $\mathit{dataEntity}$ (i.e, $\mathit{forall}\ x : \nu\ \mathit{in}\ \ell.\ p$, where ν has type $\mathit{dataEntity}$, every node in ℓ will be treated as a $\mathit{dataEntity}$). This type of semantic error can be difficult to debug.

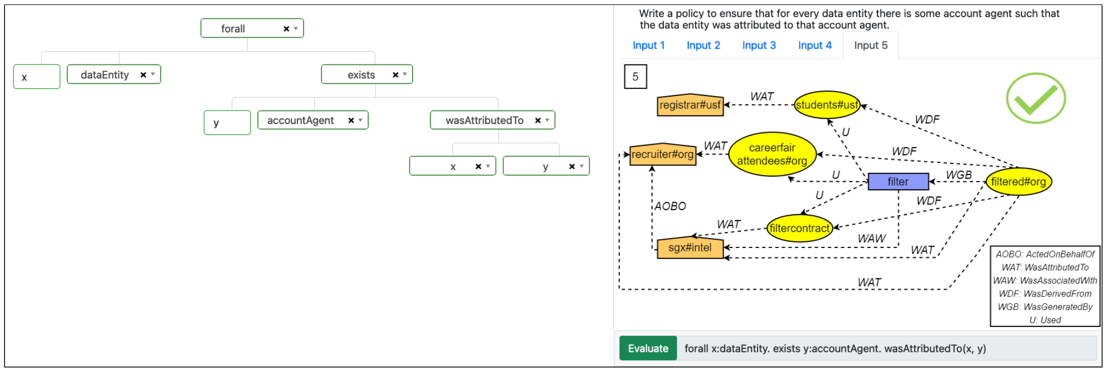
The interface helps policy writers visualize their policies and test them on a test suite before being deployed. The interface consists of a number of *selectors* that enumerate the policy constructs of the language. When a selection is made, additional selectors may appear to aid the policy writer in constructing a complete policy. For example, for the conjunction policy $p_1 \wedge p_2$, the initial screen of the interface will contain a selector listing all the different policy types in the language. The user can begin specifying this type of policy by first selecting the "conjunction" policy, after which two more selectors will appear in a tree structure to indicate to the user that the conjunction policy is made up of two policies. This selection process can continue until the user has specified a complete policy.

1) The Main Window: The main window to the ProProv interface consists of two panels as shown in Figure 2, a left panel for visualizing and constructing policies and a right panel consisting of a test suite of provenance graphs to evaluate policies on.

The left panel consists of selectors that build a tree as selections are made. The interface consists of four types of selectors: 1) policy selectors, 2) type selectors, 3) node selectors, and 4) variable selectors. The interface always begins with a policy selector containing the text `<policy>`, indicating to the user to select a policy to continue. The policy selector enumerates all of the policy constructs of the language. Once a policy is selected, more selectors may appear depending on the selection. The type selector only appears when either a universal or existential policy is selected from the previous policy selector. This selector initially contains the text `<type>` to indicate to the user to choose a node type. This selector enumerates all possible provenance node types. Node selectors appear when an edge operator is selected from the previous policy selector. This node contains the text `<node>` indicating to the user to either enter a name of a provenance node or select a variable for a previously declared node. Variables will only appear in this selector if the variable is of the right type. For example, consider the following partially constructed text-based policy:

$$
\begin{array}{l}
\mathit{forall}\ x : \ \mathit{dataEntity}.\ \mathit{exists}\ y : \ \mathit{accountAgent}. \\
\qquad\qquad \mathit{wasAttributedTo}(\texttt{<node>}, \texttt{<node>})
\end{array}
$$

For the left node selector, the selector will only provide an option to select variable x, and for the right node selector, the selector will only provide an option to select variable y. This is because variable x has type $\mathit{dataEntity}$, variable y has type $\mathit{accountAgent}$, and data entities can only be attributed to account agents. It is semantically incorrect for an account agent to be attributed to a data entity. The last selector is a variable selector and appears when either an existential or


Write a policy to ensure that for every data entity there is some account agent such that the data entity was attributed to that account agent.

```
forall x:dataEntity. exists y:accountAgent. wasAttributedTo(x, y)
```

Fig. 2. ProProv Interface Main Window

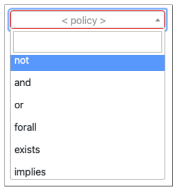


Fig. 3. Policy Selector Menu

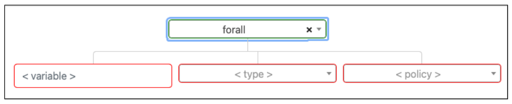


Fig. 4. Selection of a universal (forall) policy

universal policy is selected for the previous policy selector. The selector contains the text `<variable>` to indicate to the user to choose a variable name to quantify over.

The right panel consists of a suite of graphs that users can evaluate their policies on. For the study, discussed at length in Section IV, we included a test suite of graphs for the policies we asked participants to specify. Participants were provided with a test suite because: 1) during the study, the correctness of participants' policies needed to be checked automatically, and 2) the goal of this study was to measure the accuracy with which participants could specify provenance policies using the main window and not their ability to correctly construct provenance graphs. However, in practice, this panel can consist of graphical provenance nodes and arrows so that users can draw provenance graphs on which to test their policies. Alternatively, the interface can support uploading a JSON file with graph objects consisting of a list of vertices and a list of edges for users who do not want to physically draw provenance graphs.

2) Constructing Policies: To construct policies in the ProProv interface, users begin by clicking the drop down menu of the policy selector and selecting one of the listed policies, as shown in Figure 3. After making a selection, the user will continue this process for any additional selectors that may appear, until all selectors are filled.

For example, consider the policy ensuring that every data entity was attributed to an account agent, to verify that every individual who has influenced the state of data can be accounted for. Beginning with the initial policy selector, the user can select the $forall$ policy, after which three more selectors will appear: 1) a variable selector, 2) a type selector, and 3) another policy selector, as shown in Figure 4.

The user can then proceed by choosing a variable name x, $dataentity$ as a type, and $exists$ as a policy, after which three more selectors will appear for the existential policy. Next, the user can choose a variable name y, $accountAgent$ for the type, and $wasAttributedTo$ for the policy, after which two node selectors will appear. Lastly, the user can complete the policy by selecting x for the left node selector and y for the right node selector. A step-by-step illustration of the selection process for this policy can be found online [6], and the end result is shown in Figure 2.

After constructing a complete policy, users can evaluate their policies by clicking the evaluate button at the bottom of the right panel, shown in Figure 2. If the desired result is not returned, users may modify any of the selectors to create a new policy, after which they can again evaluate on the suite of example graphs provided in the right panel. The text-based version of the policy also appears at the bottom of the right panel to show users what their policy looks like in ASCII.

IV. Methodology

61 participants were recruited to measure the accuracy with which they could specify provenance policies using the ProProv interface. The usability of the ProProv interface was also measured using the System Usability Scale (SUS) [31].

A. Open Policy Agent and Rego

To provide a base for comparison, participants also completed the study using Rego, the policy specification language for the Open Policy Agent (OPA). OPA is a general-purpose policy engine which was accepted to the Cloud Native Computing Foundation in 2018. OPA can be used to enforce policies in a variety of services, including Kubernetes. OPA's popularity has increased rapidly in recent years, with a 2020 post from Styra boasting 17 million downloads and several major adopters [32].

Rego and OPA can be used to implement provenance policies and the JSON input is well-suited to representing provenance graphs. However, expressing some common policies, such as the partially completed policy in Section III-B1, requires creativity as Rego does not explicitly provide support for universal policies. Instead, such policies must be constructed by negating the opposite existential policy.

Unlike ProProv, a GUI for Rego does not exist. Instead, Rego policies are text-based and implemented using standard text editors or other textual tools such as The Rego Playground [33], an online tool for editing and evaluating policies with Rego. In contrast, ProProv was designed from the ground up with a GUI in mind, with the textual ProProv language intended only for the most advanced users.

The modified interface, which can be viewed online [6], uses the same layout as The Rego Playground with a code editor in the left panel and a panel for JSON input on the right. Like The Rego Playground, the panels use CodeMirror [34], a code editor implemented in Javascript, and the syntax highlighting for Rego developed by Styra [35]. However, the right panel cannot be edited and contains five tabs that allow participants to switch between the different inputs in the test suite. This is the same test suite provided for the ProProv interface with the graphs represented using JSON. The graphs are represented as a list of vertices and edges, with each edge defining the relation between the source and destination vertices. Upon clicking the evaluate button, the participant's code is evaluated against the test suite using OPA, and the results are returned to the participant in the same format as the ProProv interface.

B. Scenario

To increase participant engagement and create a more authentic experience, participants were provided with a detailed scenario that required the construction of provenance policies. The same scenario was used for ProProv and Rego and centered around a university career fair asking subjects to take on the role of a recruiter obtaining attendee information after the event concluded. Participants were asked to complete 7 tasks to accomplish the goal of distinguishing between students who attended the career fair and students who did not. The scenario text is provided verbatim online [6].

The seven tasks participants were asked to complete are listed below.

1) Write a policy to ensure that the data entity filtered#org was derived from data entity students#usf.
2) Write a policy to ensure that the data entity students#usf was not attributed to account agent recruiter#org.
3) Write a policy to ensure that the data entity filtered#org was derived from data entity careerfairattendees#org, and careerfairattendees#org was attributed to account agent recruiter#org.
4) Write a policy to ensure that there is some activity that used students#usf.
5) Write a policy to ensure that there is some activity that was associated with a node agent sgx#intel, and sgx#intel acted on behalf of the account agent recruiter#org.
6) Write a policy to ensure that for every data entity there is some account agent such that the data entity was attributed to that account agent.

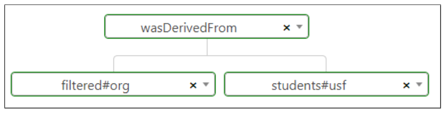


Fig. 5. The solution for implementing the first policy using the ProProv Interface

7) Write a policy to ensure that for every data entity, if filtered#org was derived from that data entity, then that data entity was attributed to either account agent recruiter#org or account agent registrar#usf.

The tasks are ordered by perceived complexity with Task 7 being considered the most difficult to construct. The perceived difficulty closely matches the actual difficulty based on participant success, discussed in Section V. Figure 5 shows the solution for Task 1 using the ProProv interface and the solution using rego can be found online [6]. The solution for Task 7 using ProProv and the solutions for Task 6 and Task 7 using Rego can be found online [6].

C. Procedure

The study was conducted in four steps: 1) demographics, 2) training, 3) policy writing, and 4) exit survey. The experiment was conducted over a 70-100 minute session. The session time was managed by the experiment site. In step one, the participants completed a standard demographics survey. Participants were also asked about their previous experience with writing policies. The demographic results are provided in Section IV-E.

Next, participants watched a training video on the ACDC provenance model. Links to each of the training videos can be found online [6]. The training focused on reading and understanding graphs as participants did not need to create their own. Participants were introduced to entities, activities, agents, and the meanings of the edge relations. The video is 7.5 minutes long.

Participants were randomly assigned to use ProProv or Rego as their first tool. The participants watched a training video on constructing provenance policies using the tool. For ProProv, participants were introduced to the various panels and selectors, and five example policies were constructed that demonstrated all of the key features. For Rego, participants were introduced to the Rego syntax and interface, and the same five examples as ProProv. The ProProv video is 8.5 minutes long, and the Rego video is 12 minutes long.

The participants were then provided the study scenario to complete with their assigned tool. Participants had access to the original training materials, which is analogous to practical situations in which developers access language documentation while constructing policies. The participant was allowed to move to the next task after constructing a passing policy or spending 2.5 minutes on the task. Participants were required to move to the next task after 5 minutes for Tasks 1-4 and 10 minutes for Tasks 5-7. Once all 7 tasks were completed, the participant repeated the process with the other tool. The maximum possible time spent is 70 minutes, however, the average was about 15 minutes for ProProv and 30 minutes for

Rego. After completing the tasks with both tools, participants completed an exit survey, provided online [6]

The survey contained 10 rating scale questions that were asked twice, once for each tool, as well as additional general questions about improving the tools or training. The results of the exit survey are discussed in Section V.

D. Initial Hypotheses

Since this is the first evaluation of ProProv, our initial hypotheses are that subject performance will be similar regardless of the tool used. Specifically, we expected two outcomes:
[H1]: Subjects would have similar failure rates when specifying policies using the ProProv interface as they do writing the policies using Rego.
[H2]: It would take subjects the same amount of time to correctly specify a policy using either tool.
We selected an alpha level of $\alpha < 0.05$ to evaluate these Hypotheses.

E. Participants

Participants were recruited from a Secure Coding course taught by study staff. The course introduced the theory of security policies and applied concepts such as the specification of firewall policies, but the course did not cover provenance policies or other concepts related to the study. Students who completed the study received extra credit for their participation. The study was approved by the university IRB, and the necessary steps were taken to prevent undue influence (e.g., the professor was not involved in recruitment or grading the extra credit, and an alternative extra-credit assignment was available). Students completed the study in an on-campus lab with a study staff member and most used their own laptop.

The study recruited a total of 61 students. All interested participants were accepted. Table I provides the demographic data. For race and ethnicity, participants could select all applicable categories, so the final totals do not sum to 61. Participants were also asked if they had constructed policies using Rego or another tool. None had used Rego before, and only eight (13%) had written policies.

V. Results

This section compares Rego and the ProProv interface using the collected data. SUS scores for the tools are also calculated based on the exit survey responses.

A. Quantitative Comparison

Table II provides a detailed overview of the data for both ProProv and Rego. The table begins with the average time spent in seconds on each task. On average, participants took about twice as much time constructing policies with Rego. To test hypothesis **[H2]**, pairwise comparisons using T-Tests were calculated for each task and the aggregate between ProProv and Rego. All of the results are statistically significant at a p-value less than 0.001, which is below the alpha level of $\alpha < 0.05$, thereby refuting hypothesis **[H2]** and allowing us to assert that subjects are able to specify policies faster with the ProProv interface.

The number of successful tasks represents the total number of participants that were able to construct a policy that passed every test case for the task. No task was passed by all participants. The hardest tasks, Task 6 and Task 7, had the lowest success rate. Only 5% of participants correctly passed Task 6 with Rego. With ProProv, 79% of participants completed Task 6, which is similar to the success rate for all tasks. The success rate for ProProv drops to 52% for Task 7. Across all tasks, participants were successful with ProProv 73% of the time and with Rego 41% of the time.

Finally, the failure rate for each task was calculated. This is the percentage of attempts that failed due to an error or an incorrect policy, calculated as the number of failed attempts divided by the total number of attempts. The aggregate failure rate for all seven tasks is also included. To test our hypothesis that subjects are equally likely to fail on a given task with either tool, a two-by-two contingency table was constructed to compare subject performance using ProProv versus Rego on each task, as well as for all tasks in aggregate. Fisher's exact test was calculated for each table. All of the results, both by each task and in aggregate, are statistically significant at an alpha-level of 0.05, thereby refuting hypothesis **[H1]** and allowing us to assert that subjects are less likely to fail when specifying a policy using the ProProv interface than Rego.

B. Usability

To measure the usability of the ProProv interface and Rego, participants were asked ten usability questions based on the SUS framework, listed online [6]. A SUS score above 68 is considered an above average score, and below 68 is considered a below average score. Scores above the average can be described as "acceptable", scores between 50 and 68 as "marginally acceptable", and "unacceptable" for scores below 50 [36]. This study also assigns an adjective (from poor to excellent) based on the SUS score [37].

The ProProv Interface scored a mean of 77.95 and median of 82.5, which is considered "excellent", and is well above the average acceptable score. Rego scored a mean of 49.59 and median of 52.5. This suggests that subjects had an easier time implementing provenance policies using the ProProv interface, which provides features specifically for constructing provenance policies.

VI. Discussion

This section discusses the implications and limitations of the study.

A. Implications

Subjects were significantly more successful in writing policies for all seven tasks using the ProProv tool. The difficulty with Task 6 and Task 7 for Rego likely spiked due to the tasks requiring features not explicitly supported by the language. Both tasks require participants to construct universal policies, which requires the participant to first create the opposite existential policy and then negate that policy. Task 7 then

TABLE I
PARTICIPANT DEMOGRAPHICS

Gender		Age		Race/Ethnicity		English Fluency		Years Programming	
Female	12 (20%)	18-24	45 (74%)	Asian	21 (34%)	Proficient	3 (5%)	None	1 (2%)
Male	49 (80%)	25-34	13 (21%)	Black	6 (10%)	Fluent	11 (18%)	Less than 2 years	14 (23%)
		35-44	2 (3%)	Hispanic or Latino	17 (28%)	Native	47 (77%)	2-5 years	41 (67%)
		45-54	1 (2%)	Other	1 (2%)			More than 5 years	5 (8%)
				White	30 (49%)				
				No Answer	4 (7%)				

TABLE II
COMPLETION STATISTICS FOR PROPROV AND REGO

	Average Time Spent (seconds)					**Successful Tasks**		**Failed Attempt Rate**		
	ProProv		**Rego**		p	**ProProv**	**Rego**	**ProProv**	**Rego**	p
	Mean	Median	Mean	Median						
Task 1	120	81	236	244	< 0.001	49 (80%)	35 (57%)	0.45	0.63	0.0371
Task 2	134	95	223	223	< 0.001	46 (75%)	34 (56%)	0.47	0.59	0.001
Task 3	135	102	240	252	< 0.001	51 (84%)	32 (52%)	0.35	0.6	< 0.0001
Task 4	147	123	220	213	< 0.001	44 (72%)	34 (56%)	0.44	0.58	< 0.0001
Task 5	204	161	314	264	< 0.001	43 (70%)	27 (44%)	0.42	0.66	0.0002
Task 6	152	103	401	401	< 0.001	48 (79%)	3 (5%)	0.38	0.96	< 0.0001
Task 7	239	189	423	450	< 0.001	32 (52%)	8 (13%)	0.58	0.9	< 0.0001
Aggregate	163	133	294	267	< 0.001	313 (73%)	173 (41%)	0.44	0.7	< 0.0001

requires participants to combine this with other features (*forall*, *implies*, and *or*) seen in the first five tasks.

Overall, subjects were correct more often than not when using the ProProv interface and were incorrect more often than not when using Rego. This is particularly insidious when taken out of the context of a (somewhat contrived) user study. When subjects create a well-formed, but incorrect, policy in this study, they are told that the policy is not correct and prompted to try again. However, in practice, an error like this represents a misconfiguration and can lead to unintended side-effects during system operation. Therefore, it is critical to minimize this failure rate, and the difference observed for the ProProv interface over Rego represents a step towards an acceptable failure threshold but requires more work and optimization of the tool to bring it in line with real system requirements.

The difference in subject timings for successful task completion indicate that subjects were significantly quicker in writing policies using the ProProv interface than using Rego, with each task taking about half as long to complete in Proprov than in Rego. This could be a combination of factors; first off, the ProProv tool allows for the quick selection of policy elements via a drop-down menu, and subjects needed only to click twice to add a policy term, whereas in Rego the subject must fully type out each term, without the aid of any features such as autocomplete, which may speed up the process. In addition, when a policy term describing a provenance relation is added in the ProProv tool, the subject is immediately prompted to fill out the elements in the relation by the tool. This is contrasted by Rego, which requires that the subjects manually type in each term, and remember the structure of a given relation, which was found to be an error-prone process.

Subjects had a much harder time using Rego, which has a broader scope with features designed for access-control policies. Note that the SUS scores for Rego in this study represent the usability when constructing provenance policies; Rego may have different SUS scores for other policies that can be constructed in Rego.

The differences in usability scoring, as well as the accuracy and speed of subject responses for the tools indicate that the ProProv tool is superior for the specification of provenance policies. There are further steps that can be taken to improve the overall usability of the ProProv tool, which presents vectors for future work and are outlined in Section VII.

B. Limitations

The subject population for this study was recruited from the student body of a large, public university. As a consequence, subjects were overwhelmingly likely (74%) to fall in the 18-24 age bracket. This does not necessarily reflect the ideal profile of a person who is likely to write policies using this tool in their professional life. This is mitigated somewhat by the subjects having several years of coding experience, with 75% of them having at least 2 years of experience writing code.

After the study was concluded, a new version of Rego was released that introduced the keyword ”every” to the language. This keyword allows for users to define universal policies without negating an existential policy. It is likely the inclusion of this operator would have improved the performance of Rego on Task 6 and Task 7, as those tasks may be defined more concisely using the ”every” keyword. The remaining five tasks are unaffected by this addition, and the results remain relevant for versions of Rego widely deployed at the time of writing.

VII. CONCLUSION

As FaaS deployments become more commonplace, more cyber analysts and end-users turn over custody of a portion of their compute stack to third parties. This makes data integrity mechanisms that can effectively operate with this reduced visibility increasingly critical to preserve data quality and confidence. The lightweight collection of data provenance and the evaluation of policies over those data products can serve as a tool to accomplish this task.

The ProProv graphical interface provides the convenience of domain-specific features for quickly and accurately defining provenance policies. When tasked with constructing provenance policies using the ProProv interface and Rego, participants were much more likely to successfully implement a policy using the ProProv interface, and did so significantly more quickly. This was especially true for tasks that required participants to combine multiple policy types or write policies that place restrictions on nodes of a particular type. Additionally, participants found ProProv to have higher usability, earning a SUS classification of "excellent" [37].

Possible directions for future work include: 1) extending the ProProv interface with advanced editing features such as real-time error checking and options for managing the visualization of the policy tree, 2) conduct studies to analyze how well users can reason about expected provenance, and 3) conduct studies of more targeted subject populations to assess the real-world needs of policy writers. Overall, the ProProv language and interface are tools that can be added to the arsenal of the integrity policy writer and help move the state of data provenance from a retroactive, forensic resource to a proactive one that can be used to ensure that data cannot be laundered by faulty or malicious services and actors.

References

[1] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring Data Integrity in Storage: Techniques and Applications," in *Proceedings of the ACM Workshop on Storage Security and Survivability*, 2005, pp. 26–36.

[2] Y. Sun, J. Zhang, Y. Xiong, and G. Zhu, "Data Security and Privacy in Cloud Computing," *International Journal of Distributed Sensor Networks*, vol. 10, no. 7, p. 190903, 2014.

[3] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2018, pp. 50–65.

[4] S. Engram, T. Kaczmarek, A. Lee, and D. Bigelow, "Proactive Provenance Policies for Automatic Cryptographic Data Centric Security," in *Provenance and Annotation of Data and Processes*. Springer, 2020, pp. 71–87.

[5] K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker *et al.*, "PROV-DM: The PROV Data Model," 2013.

[6] K. Dennis, S. Engram, T. Kaczmarek, and J. Ligatti, "ProProv Appendix," 2022. [Online]. Available: https://github.com/Ktrio3/ProProv

[7] M. Lemay, W. U. Hassan, T. Moyer, N. Schear, and W. Smith, "Automated Provenance Analytics: A Regular Grammar Based Approach with Applications in Security," in *9th USENIX Workshop on the Theory and Practice of Provenance*, 2017.

[8] R. K. Ko, M. Kirchberg, and B. S. Lee, "From System-centric to Data-centric Logging-Accountability, Trust & Security in Cloud Computing," in *Defense Science Research Conference and Expo*. IEEE, 2011, pp. 1–4.

[9] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a Timely Causality Analysis for Enterprise Security," in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[10] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime Analysis of Whole-System Provenance," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1601–1616.

[11] X. Han, T. Pasquier, and M. Seltzer, "Provenance-based Intrusion Detection: Opportunities and Challenges," in *10th USENIX Workshop on the Theory and Practice of Provenance*, 2018.

[12] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A Hybrid Approach to Enable Efficient Real-Time Provenance Based Intrusion Detection in Big Data Environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2018.

[13] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1137–1152.

[14] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[15] M. Imran, H. Hlavacs, I. U. Haq, B. Jan, F. A. Khan, and A. Ahmad, "Provenance based data integrity checking and verification in cloud environments," *Public Library of Science One*, vol. 12, no. 5, 2017.

[16] X. Han, T. Pasquier, T. Ranjan, M. Goldstein, and M. Seltzer, "FRAPpuccino: Fault-detection through Runtime Analysis of Provenance," in *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[17] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *Network and Distributed Systems Security Symposium*, 2019.

[18] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 599–613.

[20] X. Cao and L. Iverson, "Intentional Access Management: Making Access Control Usable for End-Users," in *Proceedings of the 2nd Symposium on Usable Privacy and Security*, 2006, pp. 20–31.

[21] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *4th Usenix Symposium on Internet Technologies and Systems*, 2003.

[22] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[23] E. S. Al-Shaer and H. H. Hamed, "Firewall Policy Advisor for Anomaly Discovery and Rule Editing," in *International Symposium on Integrated Network Management*. Springer, 2003, pp. 17–30.

[24] S. Brostoff, M. A. Sasse, D. Chadwick, J. Cunningham, U. Mbanaso, and S. Otenko, "'R-what?'Development of a role-based access control policy-writing tool for e-scientists," *Software: Practice and Experience*, vol. 35, no. 9, pp. 835–856, 2005.

[25] T. Tran, E. S. Al-Shaer, and R. Boutaba, "PolicyVis: Firewall Security Policy Visualization and Inspection," in *Proceedings of the Large Installation System Administration Conference*, vol. 7, 2007, pp. 1–16.

[26] S. Marouf and M. Shehab, "SEGrapher: Visualization-based SELinux Policy Analysis," in *4th Symposium on Configuration Analytics and Automation*. IEEE, 2011, pp. 1–8.

[27] D. Lomsak and J. Ligatti, "Poliseer: A tool for managing complex security policies," *Journal of Information Processing*, vol. 19, pp. 292–306, 2011.

[28] L. Bauer, J. Ligatti, and D. Walker, "Composing Security Policies with Polymer," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 305–314.

[29] L. d. Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[30] J. D. Moffett and M. S. Sloman, "Policy conflict analysis in distributed system management," *Journal of Organizational Computing and Electronic Commerce*, vol. 4, no. 1, pp. 1–22, 1994.

[31] J. Brooke, "Sus: A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, 11 1995.

[32] B. Mann, "Opa trends that prove kubernetes adoption," https://blog.styra.com/blog/opa-styra-trends-prove-kubernetes-adoption, 8 2020.

[33] Styra, "The Rego Playground," https://play.openpolicyagent.org/, accessed: 2022-02-07.

[34] CodeMirror, "CodeMirror v6," https://github.com/codemirror/CodeMirror.

[35] StyraInc, "Rego Codemirror Addons," https://github.com/StyraInc/codemirror-rego, 2019.

[36] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," *International Journal of Human–Computer Interaction*, vol. 24, no. 6, pp. 574–594, 2008. [Online]. Available: https://doi.org/10.1080/10447310802205776

[37] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *J. Usability Studies*, vol. 4, no. 3, p. 114–123, 5 2009.