WebSheets: Web Applications for Non-Programmers

Riccardo Pelizzi
Stony Brook University
r.pelizzi@gmail.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

ABSTRACT

Spreadsheets are a very successful programming paradigm. Their success stems from user's familiarity with tabular data, and their previous experience in performing manual computations on such data. Since tabular data is familiar to users in the context of web applications as well, we propose WebSheets, a new paradigm for developing web applications using a spreadsheet-like language. WebSheets can enable simple web applications to be developed without "programming," in much the same way that non-programmers create budgets or expense reports using spreadsheets. More importantly, WebSheets enable users to express fine-grained privacy policies on their data in a simple manner, thus putting them in charge of their own privacy and security concerns.

CCS Concepts

•Security and privacy → Web application security; Usability in security and privacy;

1. INTRODUCTION

Spreadsheets represent a very successful programming paradigm. Their success rests on the user's ability to create applications in a tabular format without needing traditional programming skills. In fact, non-programmers do not even think of them as applications [18, 22].

Spreadsheets were born out of necessity and convenience: people already organized data in tables and ran computations on them (e.g., financial reports) using tools such as desk calculators. A spreadsheet program greatly simplifies this task by removing most of the manual work involved in the calculation.

As far as commercial spreadsheets are concerned, the spreadsheet paradigm has remained essentially unchanged since the 1970s. In academia, most of the research on spreadsheets addressed problems from a software-engineering or programming languages perspective, to improve code reuse, correctness, and so on [6, 16, 26, 5, 13, 24]. Other works were concerned with extending spreadsheets to specialize them for a particular task, such as mashups [15, 14] or system administration [10]. In a certain sense, previous work concentrated on improving *usability*, instead of expanding their *scope of applicability*. To our knowledge, no work has been done to extend the applicability of the paradigm beyond tabular, single-user applications.

The advent of the world-wide web did not change the paradigm either: modern spreadsheets are collaborative applications in the cloud [9, 17], which should have stimulated advancements into collaboration and access control. Yet, access control mechanism have remained somewhat primitive: from the early days of Excel attachments sent around by email, to the modern days of cloud-based spreadsheets, their capabilities are essentially the same: the ability to restrict access to entire files, or individual sheets. Even if finer granularity control primitives, such as locking out individual cells from modification, were included, these were not security primitives: users can simply proceed to unlock cells and then modify them. Thus, finer granularity access control, e.g., at the granularity of individual cells or rows/columns, has not received much attention.

In this paper, *we argue that fine-grained security primitives provide the key to unlocking a whole new class of collaborative applications on the web.* We believe that our approach, based on such fine-granularity protection, represents a whole new paradigm, where data owners are empowered to create simple, customized web applications without requiring a programming background. The upside of putting data owners in charge is that they are the ones that best understand the privacy and security requirements of their data, and are hence motivated to express and enforce them through policies. In contrast, developers of web applications have other concerns — such as ease of development, ease of (or even better, absence of) configuration, and extensibility — concerns that tend to work against security, with its focus on policy configuration and access restriction.

Just as tabular computations were ubiquitous before the advent of spreadsheets and spurred the creation of the spreadsheet paradigm itself, this paper is based on a parallel realization that a lot of web pages present essentially tabular data. Applications such as calendars, event schedulers, and conference management systems present essentially tabular information. For many more applications, the HTML user interface primarily serves a cosmetic purpose, and can be stripped away to reveal a tabular representation. Ultimately, most content-based web applications store their data in database tables using a relational model, so the aforementioned stripping process could be taken to the extreme, allowing all web applications as operating on tabular data.

If tabular presentations of tabular data is common among web applications, a natural question is whether this fact can be exploited in the way they were exploited in conventional

[†]This work was supported in part by grants from NSF (CNS-0831298 and CNS-1319137) and ONR (N00014-07-1-0928).

NSPW '15, September 08 - 11, 2015, Twente, Netherlands
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3754-0/15/09...$15.00
DOI: http://dx.doi.org/10.1145/2841113.2841124

spreadsheets: *can we leverage this familiarity to enable (simple) web applications to be developed without traditional programming?* To answer this question, we first need to understand some of the reasons behind the success of conventional spreadsheets. We believe that one of the main reasons for the success of spreadsheets is their *lack of abstractions.* The internal and externally visible representations are both tabular. There are no variables, or functions as they exist in conventional programming languages; instead, spreadsheet formulas reference specific cells. Moreover, the value produced by a formula is simply thought of as a "derived cell," thus fitting within the overall paradigm of cells and tabular data.

Projecting this train of thought in the web application context, consider the popular way to design web applications, namely, the Model-View-Controller (MVC) design pattern. "Model" represents the underlying data, typically stored in a database. "View" refers to the presentation layer that is ultimately responsible for the HTML pages viewed by the user. A "Controller" updates the model or manipulates data given by the Model to generate the information to be displayed in a user-friendly manner. The core of the application logic resides within the Controller.

The main source of abstraction and trouble for a non-programmer is the split between the Model and the other two components: non-programmers can perhaps prepare HTML templates for the View, but they would have trouble manipulating the Model through the Controller by using ordinary programming languages. When viewing spreadsheets through an MVC perspective, the abstraction is not present: the cells are the Model, the View and (if they contain formulas) the Controller (the user sees these "derived cells" as a read-only part of the Model). This correspondence, along with the simple, functional flow of data in formulas is at the heart of spreadsheet's accessibility for non-programmers. Similarly, web applications with simple relationships between Model, View and Controller are amenable for development by non-programmers. The classes of web application with this property are manifold: calendars, scheduling, surveys, employment screening, budgeting, conference management, and many more.

It should be noted that when we say "non-programmers", we don't mean naive users. Just as the development of nontrivial spreadsheets requires a basic understanding of mathematical formulas, we target WebSheets at an audience that has sufficient mathematical background to understand security and privacy requirements and express them using logical formulas. We expect these users to have at least a high-school level mathematical background, including a basic knowledge of set notation. Our WebSheets paradigm is built over this knowledge.

For such users, our goal in WebSheets isn't one of outlawing all abstractions; rather, our goal is to present and use abstractions that are based on the familiar tabular format of data: we rely on named columns and rows in tables, as well as named tables. In addition, our language allows each cell to contain lists ("sets") of items, and supports simple operations for constructing new sets from existing ones.

Security and privacy concerns are paramount in a web application setting, where data belonging to multiple users are processed. In the WebSheet paradigm, users not only enter data or formulas into a WebSheet, but also *privacy policies*[1]. Such policies can be specified at the granularity of individual cells, or more commonly, at the granularity of columns, rows, or entire tables. These policies "follow" data [25, 4]: when formulas are applied over data, the output of the formula is normally subject to the same policies as the inputs.

A second important component of the WebSheets security model is the *data validation policy.* In addition to preventing data entry errors, these policies play an important role in security as well: by limiting the values that can be entered, safe behavior of operations that use this data can be ensured. For instance, consider a validation policy that requires that a reviewer select no more than specified maximum number of applications for review, and avoid reviewing applications of their students and collaborators.

If this vision were to be successful, the benefits of WebSheets are obvious: as spreadsheets ushered a new era of end-user developed applications, WebSheets will allow non-programmers to build web applications without having to be full- fledged programmers. Cutting the middle-man is not only more efficient, but also prevents a common mismatch of security concerns that is all too common when a web application is developed by a third-party: the creator of the spreadsheet and the programmer are two fundamentally different actors, and only the former understands the domain, knows which pieces of information are sensitive and can create a privacy policy. *We argue that unless the spreadsheet user can directly control the privacy policy of the data, data leaks are bound to occur.* Indeed, in today's world, web applications are developed with almost no internal security beyond the security checks scattered through the codebase: all of the web application logic is able to access all of the database, with no regard to the principle of least privilege. Invariably, this leads to massive loss of sensitive data when the web application is compromised.

In the rest of the paper, we provide an overview of the WebSheets model and language (Section 2). We then illustrate it with several example applications (Section 2). We follow with a description of the WF language (Section 3). Then, we describe a preliminary implementation (Section 4). We conclude the paper with background information on spreadsheets (Section 5) and related work (Section 6). Below, we summarize our key contributions:

- We present a new paradigm, based on spreadsheets, for non-programmers to design secure web applications.
- We present WF, a simple formula language to uniformly define operations on data, as well as privacy and security policies on the data.
- A key benefit of our approach is that the security policies are a function of the data. Such data-driven security policies are much more expressive than typical access-control policies that enforce restrictions primarily based on the user and the object being accessed.
- Another key benefit of our approach is that it enables data owners to take control of their data. It is no longer necessary to enlist an unwilling developer or administrator to understand your security requirements, and express them using the primitives provided by the underlying application. Instead, users can directly set policies, and refine

[1]In this paper, we do not make a distinction between privacy and security. Our policies can be thought of as security policies or privacy policies.

them as and when them deem necessary.

2. OVERVIEW

In this section, we introduce WebSheets. Our goal here is to present the core of the paradigm — a full-fledged system would come with more refined presentation elements, as well as a user interface. Indeed, user interfaces play a central role in spreadsheets, as most users “understand” spreadsheets in terms of their interactions with applications such as Microsoft Excel. A good user interface can further simplify a non-programmer’s tasks, e.g., most spreadsheet users don’t have to think about the fact that the cell selections that go into formulas are interpreted as relative cell addresses. However, the core system underneath the user interface knows and distinguishes between relative and absolute addresses, and indeed contains more “programming” features than what is exposed to a casual user through the interface. For similar reasons, the overview of the core WebSheets system will seem to contain more programming than what an user interface will expose to a (non-expert) user.

A focus on core capabilities also means that many presentation as well as user-interaction elements are not going to be described here. Just as presentation elements such as graphs and charts can be easily added on top of spreadsheet data, more accessible presentation elements can be implemented in WebSheets over tabular data. The same can be said about complex user-interaction elements, or control-logic. *Indeed, it is not at all our intent to outlaw programming in the context of WebSheets; rather, our goal is that programming not be mandatory for specifying and enforcing data privacy policies.* We leverage the familiar tabular data model to enable data owners to express and enforce privacy policies on their data.

Just as security policies are stated based on tabular data, user interactions such as sending of email notifications or reminders, can be provided as built-in functions. More complex control logic can also “plug into” this tabular view: such logic, potentially implemented using a conventional programming language, can consume data from one or more tables, and generate one or more tables. For instance, a visitor meeting scheduler can make use of a constraint-solving plug-in, with its input coming from tables representing availabilities of different users and their preferences. Depending on its generality, the same (or a similar) plug-in may be usable in other contexts, e.g., to assign papers to reviewers in a conference management system. Note that many of today’s web applications are based on a relational database, and hence, this view of layering control logic over tabular data is consistent with the design of today’s web applications.

Finally, there can be several choices regarding the architecture and implementation of WebSheets-based web applications. One option is to have a central server that hosts the web applications of interest to a certain user or a certain organization. In this case, all users of this server need to trust it. A second option is one of distributed implementation, where the data belonging to each user is held within an agent that enforces the user’s policies, and interacts with other agents to achieve the functionality of a WebSheets-application. Moreover, there are several choices in how much of the application logic runs within a user’s browser, and how much is located on a web server. Although all of these choices are interesting and represent research avenues on their own, we don’t explore them here. Instead, we opt for a simple, centralized implementation, as described in Section 4.

Below, we illustrate WebSheets with a few example applications. We start with a relatively simple TODO list application, and progress to a moderately complex application for faculty candidate evaluation. Through these examples, we introduce the language WF that forms the core of our approach. An interesting aspect of WebSheets illustrated in these examples is that often, security policies are as important as data transformations (“formulas”).

2.1 TODO-list

The first, simplest example is `TODO-list`, an application which maintains a private TODO list for each user. Each TODO entry has an author, a name and a “Completed” tickbox. Optionally, entries can be shared with a set of users, who can not only see the shared entry, but also check the tickbox and mark the item as completed. However, they cannot modify the name or assume authorship of the entry.

In WebSheets, this application is modeled using a single data table, plus its accompanying permission table: the `Task` table describes the data, while the `Task Permissions` table specifies the associated privacy and data validation policies. Unlike conventional spreadsheets, WebSheets applications operate on named tables; columns are also named, while rows are accessed using indexes. WebSheets cells contain expressions in a functional language called WF, which is described in Section 3. For example, cells can contain arbitrary lists. This allows convenient modeling of concepts such as associating a task with a list of users, as shown below. In addition, it is possible to represent and manipulate the content of an entire row as a WF value, and its cells can then be can then be accessed using WF’s selection operator, which is very similar to Java’s property access. For example, given a row `task` containing a column `Completed`, `task.Completed` retrieves a single cell. Finally, it is possible to represent the entire content of a table as a list of named tuples. This ability to represent entire tables means that new tables can be generated using formulas in WF. While simple applications won’t require this feature, we have found that more advanced applications can benefit from it.

Figure 1 shows the *expression view* of the `TODO-list` application. This view shows the input data (in the form of WF expressions) entered by all users and the permission table set up by the owner. The expression view is only visible by the owner of the application. The `Task` table has four columns, `Author`, `Name`, `Completed` and `Shared`. New rows can be added to the table, and they will represent new TODO items. Note that cell contents can be constants such as `"Mow Lawn"`, or lists, such as `["Frank", "Tom"]`. They can also be environment variables such as `owner`, which is bound to the user that “owns” a cell during evaluation (the user that creates a row will become the owner of all the cells in the row), `user`, which is bound to the username of whoever is evaluating the WF expression, and `this`, which is bound to the value of the current cell being evaluated.

Figure 2 shows the *value view* of `Task` for Jim. The value view is computed from the expression view, by evaluating the WF expressions of all cells into values and censoring all those values whose read permission evaluated to False. Censorship semantics aside, this behavior is similar to that of conventional spreadsheets, where the default is to show values rather than formulas. Ordinary users will interact

Task

Author	Name	Completed	Shared
owner	"Mow Lawn"	False	["Jim"]
owner	"Manscaping"	True	[]
owner	"Meet Frank"	False	["Frank", "Tom"]
owner	"Homework"	False	["Phil"]

Task Permissions

	Author	Name	Completed	Shared	All Columns
Read					user in Shared or user == owner
Write	False	user == owner	user in Shared or user == owner	user == owner	
Init	owner	""	False	[]	▨
Validate			Completed' == True or Completed' == False		▨
Add Row	▨	▨	▨	▨	
Del Row	▨	▨	▨	▨	user == owner

Figure 1: TODO List Application (Expression View)

Author	Name	Completed	Shared
"Phil"	"Mow Lawn"	False	["Jim"]
"Jim"	"Meet Frank"	False	["Frank","Tom"]
"Jim"	"Homework"	False	["Phil"]

Figure 2: Jim's Value View of `TODO-list`

with the list using this view. The content shown by the value view is user dependent: users can not only see different permissions (e.g. `user == owner` evaluates to True only if the current user is the owner of the table. When the expression is used for a read permission, it roughly translates to "only the owner of the table can see this cell"), but potentially also different values.

The functionality of `TODO-list` stems from the ability of different users to concurrently view and/or modify the list. Thus, its logic resides almost entirely in the `Task Permissions` table. Note that while data tables can have an arbitrary number of rows and columns, permission tables follow a fixed schema: there are 6 rows and $n + 1$ columns, where n is the number of columns in the data table, and the extra column is the `All Columns` column. The schema allows specifying a different policy for all cells along each column (using the first n columns), and one for all cells across every row of the table, using the `All Columns` column[2]. When permissions are specified at multiple levels of granularity, the resulting permission is the intersection of all permissions. It is helpful to think about permission tables as a tabular representation for ACLs, where columns represents sets of objects (e.g. all cells along a column) and rows represent operations.

Read permissions are specified along the `Read` row. This policy states that any user that is included in the `Shared` column of a row is permitted to view the row. In addition, row owners (i.e., users that created the row) are permitted to view the row. WebSheets will omit any row that a user cannot read. This is why Jim's view of the `Task` table shows only three tasks, while the table actually contains four rows. (In a more complex case, where some cells in a row are readable and others are not, unreadable cells will be grayed out. This, of course, reveals to the user the he/she has been denied access, and this, in itself, may leak some information. It is not our goal to eliminate all such covert channels.). Unlike all other permissions, read permissions flow together with their related cell values: the read permission of a cell does not depend only on its read permission expression(s), but also on the read permissions of all its dependencies. This allows developers to attach policies to cells and not worry about their value leaking through other cells with less restrictive policies. The evaluation and propagation of read permissions is detailed in Section 3.1.

Not surprisingly, write permissions are more tightly controlled. The policy in `Task Permissions` indicates that no one can modify the `Author` field. In addition, the `Name` and `Shared` fields can be modified only by the owner, while the `Completed` field can be modified by any one that is listed in the `Shared` column. Note that our approach provides a very simple and natural way to express access policies that are a function of data contained in the tables.

In addition to reading and writing existing cells, users can also add or delete rows from a table. Permissions for these operations are also specified in `Task Permissions`. This version of `TODO-list` permits any user to add new rows to the table, while deletion of a task is possible only by the task owner.

When users enter data into the list, they are subject to validation policies specified in `Task Permissions`. In particular, when a user adds a new row, the content of the new row are initialized using the `Init` row. When an existing row is modified, the `Validate` row specifies the validation functions that would be run on the row. In this simple example, we only ensure that `Completed` fields have boolean values.

Note that invalid fields in permission tables are grayed out. For example, add row permissions can only be defined on entire rows: defining add row permissions on a single column or init values for an entire row would not make sense.

Note that there are no formulas to deal with Denial-Of-Service attacks (e.g. adding 10000 TODO items shared with everyone, to pollute their view). In this paper we focus on the problem of data privacy, and leave out both the issue of how to protect against such attacks with the current permission table format (e.g. a count check in the `Add Row` permission) and the issue of how to extend the current permission table format with dedicated fields.

Extensions. The central benefit of WebSheets is that it enables many simple and varied customizations. This version of `TODO-list` permits any user to add new rows to the table. However, other choices, such a limiting to a specified

[2]This fixed schema does not support specifying a policy at the level of granularity of a single cell. This can be easily supported by an additional shadow table with the same number of columns and rows as the data table, but we leave it out of our description for simplicity.

Event

Author	Name	Public	Invitees	Attendees
owner	"Decimation"	False	["Crassus", "Pompey"]	Response[Name == EName and Coming == True].User

Event Permissions

	Author	Name	Public	Invitees	Attendees	All Columns
Read						user in Invitees or Public == True or user == owner
Write	False				False	user == owner
Init	owner	""	False	[]	Response[Name == EName and Coming == True].User	▨
Validate			this or Invite != []			▨
Add Row	▨	▨	▨	▨	▨	
Del Row	▨	▨	▨	▨	▨	user == owner

Response

User	EName	Coming
owner	"Decimation"	True
owner	"Decimation"	False

Response Permissions

	User	EName	Coming	All Columns
Read				user in Event[Name==EName].Invitees or Event[Name==EName].Public or user == owner
Write	False			
Init	owner	""	False	▨
Validate		user in Event[Name==EName'].Invitees or Event[Name==EName'].Public or user == owner		▨
Add Row	▨	▨	▨	
Del Row	▨	▨	▨	user == owner

Figure 3: RSVP Application (Expression View)

Event

Author	Name	Public	Invitees	Attendees
"Caesar"	"Decimation"	False	["Crassus", "Pompey"]	["Crassus"]

Response

User	EName	Coming
"Crassus"	"Decimation"	True
"Pompey"	"Decimation"	False

Figure 4: Caesar's Value View of the RSVP application

list of users, is also possible. Note that such a list can also be specified as another table, e.g., we could call it Task Users. This list may be defined and modified in the same manner as the Task table itself, thus providing another interesting application of data-based security policies. In this simple example, no validation policies are included. However, we may want to include a policy that states that task status can be modified from False to True, but not the other way around.

Another possible extension is to add columns to indicate whether a payment was made for a task (Paid), and a column to indicate acknowledgment of payment (Received). This extension also needs another column, say, Selected, to indicate the specific person from the Shared list that selected and completed the job. Permissions would now be modified so that only the person in the Selected column can modify the Received column. In addition, row deletion would be prohibited until the Received column is true.

2.2 Event RSVP

The second example is an Event invitation and RSVP web application, RSVP. Users can either create public or private events, and can RSVP to other user's events. However, users can only RSVP to private events they are invited to.

This application requires two data tables, Event and Response. The Event table has the Author, Name, Public, Invitees and Attendees columns. We omit other data such as location, date, reason for refusal, etc to simplify the example. The Attendees column is dynamically calculated from the set of users who RSVPed using the Response Table, which has the Author, EName and Coming columns. Figure 3 shows some initial data for one event and two responses, plus the permissions required to implement the access control policy, while Figure 4 shows Caesar's view of the web application, the author of the sole event.

The permission metadata for Event uses formulas similar to the TODO-List example: the Author column is fixed to the author of the event with the same non-writable init value owner. The Attendees column, which is fixed to a static formula, uses list filtering: it returns the authors of all positive responses from the Response table that have the same event name. Note how the cells are directly addressable by their column names inside the brackets. While programmers are familiar with the idea of working with additional variable bindings implicitly introduced in an inner scope, it may appear that this notation is too complex for non-programmers. However, note that the filtering expression is no different from set construction notation that is familiar in high-school mathematics:

$$\{x | x \in Z \text{ and } x \text{ is a multiple of two }\}$$

The primary difference is that the selector expression in WF can make use of column names. We argue that while this generalization may need an introduction, it is not too difficult to master. An appropriate user interface can further simplify this task by providing a simple way to construct these expressions.

The `Read` metadata enforces that events can only be seen by the author, an invitee or, if the event is public, by everyone. The `Write` row uses permissions from different levels of granularity: the `All Columns` metadata only allows the author of a row to edit any of its cells, while the `Author` and `RSVP` columns further restrict the writable fields to `Name`, `Public` and `Invitees`. The `Validation` metadata specifies that an event should either be public or have at least one invitee. Note how in the validation formula `Public` is also bound as `this` because the formula is evaluated in the context of a particular cell and is provided with bindings to the current cell, row, column etc. `Add Row` and `Del Row` have the same logic as the `TODO-List` example: anyone can create events, but only the owner can delete them.

The permission metadata for `Response` involves more complex formulas. The `Read` row mandates that users can only see responses if they are the author of the response, if they are invited or if the event is public. The `Validate` row and the empty `Add Row` row specify that users can create new empty responses, but can only set the `EName` cell (and thus link their response to a specific event) to events that are either public or to which they have been invited.

2.3 Faculty Candidate Review

To further illustrate the applicability of WebSheets using a more involved example, we employ a scenario from our experience in academia: admittance of new faculty into the department. The admittance process is by nature collaborative, since existing faculty can contribute to the process: each candidate presents his research to the faculty body, who can later grade the candidate. Finally, faculty picks the best candidate, using the grades provided. However, because professors are too busy or not familiar with web development, our department currently passes around a spreadsheet file by email to be filled out and returned to the department chair. As the size of the faculty keeps increasing, this process gets more and more unwieldy. The workflow is far from optimal not only because of the lack of automation (how to merge concurrent changes?), but also because there is no access control (what about conflicts of interests?). WebSheets can be employed not only to automate the grading process, but also to enforce security policies. In particular, we seek to enforce the following properties:

1. Applicants can only enter and view their own application, while Faculty can view any application.
2. Each Faculty can enter at most one review per candidate, and cannot enter a review for another Faculty.
3. To avoid being influenced by others, faculty should not see each other's grades for a particular student until they have graded the student themselves.
4. If the candidate and one faculty have a conflict of interest, the latter should be barred from grading and seeing the applicant's grades and average.

BestApplicant

Name	Average
`((Name=a.Name,Average=TRUST(a.Average)))` `for a in Applicant when TRUST(a.Average)>3.5)`	

Figure 6: Dynamic Table with declassified averages

This WebSheet, shown in Figure 5, requires 3 tables, `Faculty`, `Application` and `Review`. The first table `Faculty` is used to identify which users have faculty privileges. In this simplified example, merely having an entry in the table grants the user faculty privileges; any other user is assumed to be an applicant. A more involved example would use additional fields to assign roles to users. Read permissions are blank, because the faculty roster is public; write permissions on the rows are set to `False`, because we assume that the Faculty roster has been pre-populated by the WebSheets administrator.

The `Application` table has the columns `Name`, `Conflicts`, `AppReviews` and `Average`. This table is used by applicants to fill in their application: each applicant fills in the first two fields, while the latter two fields are dynamically calculated by read- only formulas which use the reviews submitted by faculty. The `Read` formula for all the columns asserts that applications are only visible by the applicant or by a member of the faculty body. Note how no `Read` restrictions are required for `Grades` and `Average` to prevent faculty with a conflict to view the applicant's grades: because permissions follow data, it is sufficient to predicate on the grades themselves, which is done in the permission table for `Review`. Any grade that violates the aforementioned security properties would a) be omitted from the list of grades, and b) prevent the faculty from seeing the average. Once all applicants have been graded, the faculty body sorts this table and examine the top applicants to select the most suitable faculty candidate.

The `Review` table has the columns `Author`, `AppName` and `Grade`. This table is filled out by faculty after they have evaluated the applicants. The `Read` permission for all columns restricts read access to faculty only. The Grade `Read` permission predicates that to view a grade, a user must either be the author of the grade or must have graded the same applicant already, and should not have a conflict with the applicant. Note that the `Read` policy from the `All Columns` field still affects this field, and the resulting policy is the intersection of the two policies. The `Add Row` permission specifies that only faculty can write new reviews. This permission works together with the `All Columns Write` permission, which restricts review edits to the author of the review. The `Validate` check for the `AppName` field restricts which values it can contain: it specifies that the new value entered into the cell should not be a) the name of an applicant that has a conflict with the current faculty and b) the name of an applicant already reviewed by the current faculty. The `AppName'` variable is bound to the new value of `AppName` which will be inserted into the table if the write check succeeds.

Figure 2.3 uses a slightly different interpretation of the properties above to present two additional features, namely dynamic tables and declassification: while individual reviews are still hidden from faculty in case of conflicts, we want to allow all faculty to view averages.

To show averages to all faculty, we leverage dynamic tables to generate an additional `BestApplicant` table, which will

Faculty

Name
"Bell"
"Murphy"

Faculty Permissions

	Name	All Columns
Read		
Write		False
Init	""	▨
Validate		▨
Add Row	▨	False
Del Row	▨	False

Applicant

Name	Conflicts	AppReviews	Average
owner	["Bell", "Murphy"]	Review[AppName==Name].Grade	AVG(AppReviews)
owner	[]	Review[AppName==Name].Grade	AVG(AppReviews)

Applicant Permissions

	Name	Conflicts	AppReviews	Average	All Columns
Read					user == owner or user in Faculty.Name
Write	False		False	False	user == owner
Init	owner	[]	Review[AppName=Name].Grade	AVG(AppReviews)	▨
Validate					▨
Add Row	▨	▨	▨	▨	
Del Row	▨	▨	▨	▨	user == owner

Review

Author	AppName	Grade
owner	"Smith"	4
owner	"Doe"	3.5

Review Permissions

	Author	AppName	Grade	All Columns
Read			user == owner or user in Review[AppName==row.AppName].Author and user not in Applicant[Name==AppName].Conflicts	user in Faculty.Name
Write	False			user == owner
Init	owner	""	0	▨
Validate		user not in Applicant[Name==AppName’].Conflicts and user not in Review[AppName==row.AppName].Author		▨
Add Row	▨	▨	▨	user in Faculty.Name
Del Row	▨	▨	▨	user == owner

Figure 5: Faculty Admission Application (Expression View)

list all applicants with a good average grade. Note that `BestApplicant`, being a dynamic table, has no permission information of its own: the permissions of its dynamic content depend on the permissions of the data that is being used to generate the table. The table builds its contents from a single WF expression, in this case a list construction formula. The result of evaluating the dynamic table expression must always be a list of tuples with the same keys, which is used to fill out the rows and columns of the resulting table. From the user's point of view, a dynamic table is accessed transparently through rows and columns like an ordinary static table, except that its fields are read-only and cannot be evaluated individually, because their value depends on one single formula. This particular table uses the intermediate results from `Applicants`'s column `Reviews` to simplify the calculation of the average.

Note that here we use the privileged function `TRUST`, because faculty members need to see the average even in case of conflicts. In a sense, we trust the `AVG` function to have properly anonymized the confidential data. We point out that, although this seems to weaken the security of WebSheets, the state of the art in textual web applications is that every single operation is trusted and has full privileges by default, while here it requires a special construct that can be used sparingly.

3. THE WF LANGUAGE

WebSheets cell and permission formulas are written in WF, a simple, functional, pure, strongly-typed language. The target audience of the language is the same non-programmers who are able to write Excel formulas.

Figure 7 shows the grammar for the language, which starts from the nonterminal `<expr>`. `INT`, `FLOAT`, `BOOL`, `STRING` and `ID` are returned by the lexer.

```
<expr> ->
   INT
 | FLOAT
 | BOOL
 | STRING
 | ID
 | '[' (<expr> (',' <expr>)*)* ']' ## list
 | '(' ID '=' <expr> (',' ID '=' <expr>)* ')'
     ## named tuple
 | <expr> <bin_op> <expr>
 | <expr <un_op> <expr>
 | '(' <expr> ')'
 | <expr> '.' <expr> ## selection
 | <expr> '{' <expr> (',' <expr>)* '}' ## projection
 | ID '(' (<expr> (',' <expr>)*)* ')' # function call
 | 'if' <expr> 'then' <expr> 'else' <expr>
 | '(' <expr> 'for' ID 'in'
     <expr> (',' ID 'in' <expr>)* ['when' <expr> ')']
       ## list construction
 | <expr> '[' <expr> ']' ## list filtering
<binop> ->
 <expr> '+' <expr>
 ## ... standard math and logic operators ...
 | <expr> '++' <expr> ## list & tuple concatenation
 | <expr> 'in' <expr>
 | <expr 'not in' <expr>
<unop> ->
 'not' <expr>
 | '-' <expr>
```

Figure 7: EBNF Grammar for the WF Language

Although the formulas are statically analyzed for dependency calculation, the evaluation is entirely dynamic, i.e., type errors or unbound identifiers will only cause an error at runtime. Evaluation will turn WF expressions into WF values, potentially requiring evaluation of other dependent cells in the spreadsheet. Identifiers are first looked up in the environment, a context- dependent `(name, value)` map containing information such as the current user, the current table or the current column (when applicable). When an identifier is not found in an environment, the name is assumed to be table name, which can be further refined to select a particular row or column.

Besides the usual scalar data types (`Int`, `Float`, `Bool`, `String`), it supports two composite types:

- *Lists*: A (possibly empty) ordered collection of WF expressions, Lists can be concatenated, sliced and accessed with the operations defined below.
- *Named Tuples*: A (possibly empty) unordered `(key, value)` map. Tuples can also be merged, sliced or accessed.

Lists and Named Tuples are not provided just to express application logic: WF's semantics define a dualism between table operations and WF values. At runtime, a WebSheets table is represented as a list of named tuples: each element of the WF list represents a table row, and each row is represented as a WF tuple, a map of column names to WF values for that particular row.

Besides the usual unary and binary operators from math and logic, manipulation of data in WF is supported by four main constructs, which also have a dualism with table operations:

- *Selection*: access a single element from a list (`[5,6,7].1 -> 6`), a single key-value from a named tuple (`(a=1,b=2).a -> 1`), or perform tuple selection on all elements of a list of named tuples (`[(a=1,b=2),(a=3,b=2)].a -> [1,3]`). This is the WF equivalent of selecting one particular row, cell or column of a table respectively.
- *Projection*: returns a subset of elements from a list (`[5,6,7]{0,1} -> [5,6]`), multiple key-values from a named tuple (`(a=1,b=2,c=3){a,b} -> (a=1,b=2)`), or multiple columns from a list of named tuples: (`[(a=1,b=2,c=3),(a=3,b=2,c=5)].{a,c} -> [(a=1,c=3),(a=3,c=5)]`). This is the WF equivalent of selecting multiple rows, cells or columns of a table respectively.
- *List Construction*: combines one or more lists to create a new list. For example:

```
  ((a=b, c=d) for b in [1,2], d in [3,4]
    when b+d>4)
  ->  [(a=1,b=4), (a=2,b=3), (a=2,b=4)]
```

  This is the most general operator in the language. The expression `(a=b,c=d)` is evaluated using different bindings for `b` and `d` for each iteration. For example, the environment for the first iteration binds `b` to 1 and `d` to 3. The set of bindings to iterate on represents the Cartesian product of the lists supplied, and the result is a list containing all the evaluations of `(a=b, c=d)` using all the bindings from the Cartesian product. If expression returns a tuple, then this is the WF equivalent of building a new table from existing tables. The `when` clause can be used to filter rows from the resulting table. In this case, the first binding does not appear in the result because it is filtered out by the clause. Many languages have an equivalent construct called list comprehension. Functional programmers will note how it performs both `map` and `filter`.
- *List filtering*: returns the subsets of elements that satisfy the condition in brackets (`[(a=1,b=2)][a-b>0] -> []`). The construct evaluates the condition in the context of each element, and only includes the element in the result if the condition evaluates to True. Note how the keys of the tuple are automatically added to the environment for each element. List filtering is particularly useful when the WF list is actually a WebSheets table: `Table[cond]` effectively performs table filtering, returning a subset of `Table` whose rows all satisfy the condition `cond`. List filtering is a shorthand for the list construction operator: `e[c] == (v for v in e when c)`, except that list construction does not automatically add the keys of `v` to the environment upon evaluation of `c`.

3.1 Values and Read Permissions

Evaluation of formulas is interleaved with evaluation and propagation of read permissions (other permissions do not need to be propagated and can be evaluated as needed). Intuitively, the permission propagation semantics are similar to taint-tracking semantics [11] for ordinary operations (e.g. addition: `perm(a+b) = perm(a) && perm(b)`. Potential falsy values come from the evaluation of cells, either indirectly because the cell's value formula depends on an unreadable cell, or directly because their permission formula does not evaluate to True. The following formula informally captures the semantics for a cell `cell` in a table `t`, row `r` and column `c`:

```
perm(cell) =
  eval(cell).perm &&
  eval(perm_row(t,r)) &&
  eval(perm_col(t,c))
```

This formula only considers permissions at row and column granularity, but can be extended to support other levels (e.g. `perm_cell(t,r,c)` for cell-level granularity).

WF values store their read permission along with their value, except for Lists and Named tuples: operation on these container types maintain information about the permission of each element. This allows converting tables into lists of named tuples back and forth without losing any permission information. This is especially important to support dynamic tables, which are compressed into a single WF value (which must be a list of named tuples) before they are spread over multiple fields.

4. IMPLEMENTATION

We are implementing WebSheets as web application which communicates with a backend server written in Haskell through a JSON API. Through the web application, users use the backend as a central repository of applications to collaborate with other users.

The implementation of the backend server is currently in the advanced prototype stage. Most essential features have been completed. In particular, the following features have been implemented:

- an LR *parser for the WF language*, based on the grammar shown in Figure 7.
- *static dependency analysis* for WF formulas, to build a support graph and recalculate a minimal amount of cells when formulas change.
- support for both *static tables* and *dynamic tables*, as described in Section 2.
- *evaluation* of data and permissions according to the semantics informally specified in Section 3.
- *access control* on the backend operations, including censorship of values with invalid permissions.

On the other hand, some features have not yet been implemented:

- *no concurrency*: concurrent reads are trivial to implement, but the implementation of concurrent writes requires more care.
- *incomplete API*: several API operations have not been implemented yet. For example, it is currently not possible to edit table permissions at runtime. For the time being, we rely on importing table pre-filled with permission formulas from XLS files, with a format similar to the examples seen in Section 2.
- *HTML interface under development*: the WebSheets codebase supports two different interfaces: a textual, command-line interface and a web interface using a JSON API. Currently, only the former is fully functional.

5. BACKGROUND

VisiCalc [1] introduced spreadsheets in their modern form. Since then, spreadsheets have enjoyed tremendous commercial popularity. Unarguably, their main contribution to modern computing and the reason for their success is that they enabled non-programmers (accountants, administrators, secretaries, etc) to enter, process and visualize data in a tabular format [18, 22], effectively allowing them to create full-fledged data-driven applications. The major testament to their ease-of-use is that users rarely refer to their spreadsheets as applications.

The success of spreadsheets can be traced to a handful of key features:

- *User Interface*: while textual programs are developed by editing text files which operate on abstract data structures, spreadsheets programs enable users to organize their data visually into concrete rows, columns and tables.
- *Concrete Programming*: because spreadsheet formulas can only be used to calculate a single cell value, the non-programmer is not bogged down by abstractions: the relationships that make up formulas are always about cells instead of columns, rows and tables.
- *Instant feedback*: while textual programs must be restarted (and possibly recompiled) after their code is changed, spreadsheets have update semantics that only recalculate cells as needed, returning an updated state almost instantly.

As far as commercial spreadsheets are concerned, the spreadsheet paradigm has remained fundamentally the same since the 1970s, and researchers have eloquently argued their weaknesses [3, 7]:

- *Code Duplication*: the lack of abstraction mentioned above is mitigated through generous use of copy & paste. For example, to sum the contents of two columns of size n in a third column, because of the lack of a `map` primitive, the formula for `C1 =A1+B1` is copied n times all the way to `Cn`. When the formula is updated (e.g due to a bug) the user must find and update all copies manually.
- *Brittle References*: There are two sources of brittleness. Firstly, most spreadsheet products handle the aforementioned copy & paste by interpreting A1 and B1 as relative references, which become `Ai` and `Bi` respectively when copied into the i-th row. Although convenient, the semantics are confusing to users and are a source of bugs. Secondly, because tables can share worksheets, the addition of rows and columns can often move the cell referenced, causing the original formula to refer to a different cell.
- *Formulas and Macros*: The formula language is very simplistic, and many operations can only be achieved using macros (e.g. define a reusable function). However, macros can only be developed by programmers, which defeats the purpose of using a spreadsheet for most users.

WebSheets preserve all the benefits of the basic spreadsheet model, while mitigating some of its drawbacks: fixed table schemas with column names reduce the brittleness of formulas, while the WF language reduces the need for a general-purpose imperative language for macros. While this paper focuses on the advantages of using the familiar spreadsheet model for the development of web applications, future work will investigate further on how to overcome the limitations imposed by the model. Solutions might be non-technical in nature and rely on user training: Reference [19] presents a meta-analysis on spreadsheet errors, and describes how software engineering advancements have yet to make their way into spreadsheet development.

6. RELATED WORK

Although the programming languages community in general has largely ignored spreadsheets [3], the functional programming community has attempted to improve the paradigm. Functional programmers have a closer relationship with spreadsheets, mainly because spreadsheet computation is a form of pure functional computation [22].

The related work in this area falls roughly into three categories: extension and specialization of the paradigm for a specific use case (e.g. web scraping), backwards-compatible or user-friendly improvements that cater to non-programmers (e.g. define functions in excel formulas), and backwards- incompatible improvements that cater to programmers (e.g. replace Excel's language with a full-fledged functional language).

Domain-Specific Spreadsheets: this research area is motivated by the idea that spreadsheets are an incredibly productive tool, but cannot be used in specific domains because they lack necessary features. Vegemite [15] and Reference [14] focus on importing and interacting with web data within the spreadsheet paradigm. The former runs within the browser to closely interact with web pages and offer macro recording capabilities, to avoid repetitive tasks and define web scraping macros, while the latter is an Excel add-on focused on pulling data from a multitude of web services. A1 [10] is a spreadsheet tool to simplify system administration. Unlike traditional spreadsheets, cells can contain Java Objects and other cells can invoke these objects to display the result of a computation. The paper also introduces the concept of events, making the spreadsheet reactive in the face of updates, timer intervals, etc. XCelLog [23] introduces *Deductive Spreadsheets*, namely spreadsheets augmented with a DataLog engine for policy development. With its instant updates and tabular representation, the authors argue that spreadsheets are a great fit for exploratory policy development.

Spreadsheets for Non-Programmers: this research area focuses on improving traditional spreadsheets while maintaining their ease-of-use. Ideally, the changes should be backwards-compatible and introduced gradually to the user. Reference [13] introduces reusable functions written in tabular form into Excel: currently, users must abandon the spreadsheet paradigm and resort to Visual Basic macros to define even the simples of functions. Tabular functions use cells for input, output and intermediate computation and allow the user to define functions completely within the spreadsheet paradigm.

Spreadsheets for Programmers: these works attempt to repackage the spreadsheet paradigm with programmer-friendly features. Reference [24] augments Excel with the possibility of calling externally defined Haskell Functions by communicating with an external Haskell interpreter. Reference [6] details the implementation of a Spreadsheet Engine and UI using the Clean Language, a lazy functional language. Haxcel [16] presents a Spreadsheet-like interface for Haskell development. Reference [26] describes Mini-SP, a language for spreadsheets focused on supporting non- trivial control flow and message passing among cells. Reference [5] discusses how functional and object-oriented features can come together in the spreadsheet paradigm.

Overall, none of the papers discussed above are similar to WebSheets, because they focus on improving spreadsheet *usability* (for a particular task, for non-programmers and for programmers respectively), while WebSheets expand the *applicability* of the spreadsheet paradigm to an entire new class of applications, namely web applications.

Web Application Policies Another area of related work is about retrofitting a more principled approach to security in the textual web applications paradigm.

GuardRails [2] discusses modifying Ruby On Rails (RoR), a popular MVC Ruby Web Application Framework, and provides a source-to-source translator to add data-centric policy enforcement in exiting RoR applications. Since the Model in RoR is always accessed and manipulated through `ActiveRecord` objects (Object-Oriented interfaces to the underlying database), GuardRails allows attaching policies to these objects, so that policy checks can be enforced by the framework itself when application code operates on the objects. This separates the application logic from the security policies.

Resin [25] also separates application logic from security policies, but in a more generic way. It defines *policy objects* (e.g. only readable by admin) and *filters* (e.g. HTML output). Policy objects are attached to data; Resin modifies the runtime to propagate policy objects along with data; when data with a policy objects passes through a filter, the filter and the policy object work together to perform a policy check. The combination of policy objects, data tracking and filters have similar capabilities to information-flow control (IFC). Aeolus [4] and SAFEWEB [12] have similar goals, but apply more familiar IFC concepts, propagating *confidentiality* and *integrity* labels. Besides performing IFC on the server, Hails [8] also enforces privacy policies on the browser against data leaks. Reference [20] also performs IFC, but uses Erlang message-passing capabilities and lightweight threads with an actor-based architecture to enforce isolation and label tracking at the language level.

CLAMP [21] is a modification to the traditional LAMP stack that enforces strong isolation between users, to limit the damage from not only missing checks, but also malicious code. The architecture uses Virtual Machines to serve each user from a dedicated server, and a SQL proxy enforces a different policy for each user.

The main contribution of these works in relation to WebSheets is the realization that current security practices in web applications have much to be desired, which stimulated us to seek a more principled approach from the ground up.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented WebSheets, an extension to the spreadsheet paradigm to allow non-programmers to build web applications. We extended traditional spreadsheets with data-centric policies expressed through metadata tables and presented WF, a simple functional language tailored to non-programmers.

The paper uses 3 examples to claim that a) many web applications can be easily expressed with this paradigm, and b) that policies can be easily developed by non-programmers. Given that the prototype and the examples presented are small scale, we cannot conclusively establish claim b). However, this remains one of our main motivations and one of the primary directions for future work.

What this paper shows is that for simple web applications, policies are not overly complex, and more importantly, the functionality of the application is a direct consequence of the policy.

8. REFERENCES

[1] D. Bricklin. VisiCalc: Information from its creators. http://www.bricklin.com/visicalc.htm, 2014.

[2] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: a data-centric web application security framework. In *USENIX WebApps*, 2011.

[3] R. J. Casimir. Real programmers don't use spreadsheets. *ACM SIGPLAN '92*.

[4] W. Cheng, D. R. Ports, D. A. Schultz, V. Popic,

A. Blankstein, J. A. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in aeolus. In *USENIX ATC*, 2012.

[5] C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Glasgow Workshop on Functional Programming*, 1997.

[6] W. A. De Hoon, L. M. Rutten, and M. C. D. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 1995.

[7] C. H. Q. Forster. Programming through spreadsheets and tabular abstractions. *J. UCS*, 2007.

[8] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[9] Google. Google Sheets. https://www.google.com/sheets/about/, 2015.

[10] E. M. Haber, E. Kandogan, A. Cypher, P. P. Maglio, and R. Barrett. A1: Spreadsheet-based scripting for developing web tools. In *LISA*, 2005.

[11] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF 2012*.

[12] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. Safeweb: A middleware for securing ruby-based web applications. In *Proceedings of the 12th International Middleware Conference*, 2011.

[13] S. P. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in excel. In *ACM SIGPLAN*, 2003.

[14] W. Kongdenfha, B. Benatallah, J. Vayssière, R. Saint-Paul, and F. Casati. Rapid development of spreadsheet-based web mashups. In *WWW '09*.

[15] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *International conference on Intelligent user interfaces*. ACM, 2009.

[16] B. Lisper and J. Malmström. Haxcel: A spreadsheet interface to haskell. In *Workshop on the Implementation of Functional Languages*, 2002.

[17] Microsoft. Office 365. https://products.office.com/en-US/, 2015.

[18] B. A. Nardi and J. R. Miller. *The spreadsheet interface: A basis for end user programming.* Hewlett-Packard Laboratories, 1990.

[19] R. R. Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing*, 1998.

[20] I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. Enforcing user privacy in web applications using erlang. *W2SP, Oakland, CA*, 2010.

[21] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *IEEE S&P*, 2009.

[22] P. Sestoft. Implementing function spreadsheets. In *ACM Workshop on End-user software engineering*, 2008.

[23] A. Singh, C. Ramakrishnan, I. Ramakrishnan, S. D. Stoller, and D. S. Warren. Security policy analysis using deductive spreadsheets. In *ACM workshop on Formal methods in security engineering*, 2007.

[24] D. Wakeling. Spreadsheet functional programming. *Journal of Functional Programming*, 2007.

[25] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM SIGOPS*, 2009.

[26] A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. In *International Conference on Computer Languages*. IEEE, 1994.