End-User Regression Testing for Privacy

Swapneel Sheth, Gail Kaiser
Department of Computer Science, Columbia University, New York, NY 10027
{swapneel, kaiser}@cs.columbia.edu

***Abstract*—Privacy in social computing systems has become a major concern. End-users of such systems find it increasingly hard to understand complex privacy settings. As software evolves over time, this might introduce bugs that breach users' privacy. Further, there might be system-wide policy changes that could change users' settings to be more or less private than before.**

We present a novel technique that can be used by *end-users* for detecting changes in privacy, *i.e.,* regression testing for privacy. Using a social approach for detecting privacy bugs, we present two prototype tools. Our evaluation shows the feasibility and utility of our approach for detecting privacy bugs. We highlight two interesting case studies on the bugs that were discovered using our tools. To the best of our knowledge, this is the first technique that leverages regression testing for detecting privacy bugs from an end-user perspective.

***Index Terms*—Social Testing; Privacy; Regression Testing;**

I. Introduction

Today's college students do not remember when websites dedicated to or containing social networking aspects, such as Facebook, Twitter, Amazon, Netflix, Last.fm, and StumbleUpon, were not commonplace. Privacy in the context of these social computing systems (henceforth, "social system") has become a major concern for the society at large. A search for the pair of terms "facebook" and "privacy" gives nearly two billion hits on popular search engines. With many online systems that range from purchasing products to recommending movies to watch, as well as media attention (*e.g.,* the AOL anonymity-breaking incident reported by the New York Times [1]), both users of the systems and even non-users of the systems (*e.g.,* friends, family, co-workers, etc. mentioned or photographed by users) are growing more and more concerned about their personal privacy [2]. In some of these cases, figuring out the privacy settings is so complicated that there exist many third-party "how to" guides [3], [4]. Recent feature enhancements and policy changes in social networking and recommender applications – as well as their increasingly common use – have exacerbated this issue [5]–[8].

End-user Software Engineering (EUSE) is becoming an increasingly important as "computer programming, almost as much as computer use, is becoming a widespread, pervasive practice." [9]. EUSE ranges from requirements and design to testing and debugging. End-user testing, in particular, is important for privacy because the end-users have little or no say in the functional specifications of or changes to social computing software, and because its online software they cannot avoid upgrading after each change or continue to use an "old version." Plus well-known social computing systems have an established history of making changes that breach privacy with no a priori ability for end-users to opt out [6]. But the end-users are not, in general, trained software engineers so any methodology and technology must be simple and easy to use without training.

Consider the following scenario for Pete – a user of a social system like Facebook. Pete is comfortable using websites and computers, but doesn't have a very strong technical background in Computer Science or Software Engineering. He is worried about his privacy when he uses Facebook though. There has been a lot of media coverage about privacy concerns, how they keep changing their privacy policy periodically, how hard it is to figure out all the privacy settings, and so on and this has caused Pete some concern. Pete likes using the system to keep in touch with his friends and professional colleagues, but he doesn't want strangers to have access to his personal information, photos, likes, dislikes, etc. He has used some of the "how to" guides to configure his settings to what he wants to them (or so he thinks).

A scenario like this raises a number of interesting software engineering research challenges:

1) **R1**: Users' Mental Model of Privacy — How can we make complex privacy settings easier to understand and verify for Pete? (*e.g.,* If I think my photos are shared with only my friends, is that really the case?) — **Requirements Engineering for Privacy.**
2) **R2**: Code/API Bugs — How can we detect if privacy settings that are in place remain the same as the software evolves and changes over time? (*e.g.,* If my photos are currently only shared with my friends, how do I know that they won't "automatically" get shared with everyone due to a software bug?) — **Regression Testing for Privacy.**
3) **R3**: Policy Changes — How can we detect system wide policy changes that might cause privacy settings to change? (*e.g.,* If my photos are private right now, how do we detect if there a policy change that makes all photos publicly accessible?) — **Regression Testing for Privacy.**

Ideally, for users like Pete who do not have access to the source code of systems like Facebook, we want to do this from an *end-user* perspective. In this paper, we present a novel technique that leverages a social, crowdsourced approach for detecting bugs from the end-user perspective. To the best of our knowledge, this is the first technique that leverages regression testing for detecting privacy bugs.

Continuing the scenario above – consider Roger, a friend of Pete on the social system. Roger can manually monitor what

part of Pete's information is visible to him. This monitoring can be done periodically as often as Pete/Roger deem necessary – say, every day, every hour, once a month, and so on.

Using this monitoring, Roger can inform Pete when the information he sees changes. For example, he might suddenly see a whole lot of new information that is now visible. This might be due to: (1) Pete added more information manually; (2) Pete changed his privacy settings either deliberately or accidentally; (3) Pete didn't do anything – there is a bug in the code or API, possibly due to code changes; and (4) Pete didn't do anything – the social system made a wide policy change where this information for many or all of its users is now visible. Pete, now, using this feedback from Roger, can decide whether it's ok for the new information to be visible and take the appropriate actions such as changing the settings back to what he wants them to do, reviewing the privacy settings or doing nothing.

This, however, can be very tedious for Roger to have to do all this manually, particularly, if frequently done, and he could easily forget to check certain things. Thus, automated monitoring is essential and can be done if the system provides an API. A lot of social systems like Facebook [10], Twitter [11], Last.fm [12], and Google+ [13] do provide an API whose main purpose is to build an ecosystem of app developers for the system. We can leverage such APIs where possible and if an API doesn't exist, the same goal can be achieved via screen scraping.

This is our broad approach — using one's friends for detecting potential privacy violations. We call this ***Social Testing***. There are more specific details that need to be dealt with based on the platform and API and we discuss this in Sections III and IV. This latter section also contains details about the feasibility of this approach and the kinds of privacy bugs it has helped us uncover. The main advantages of this approach are:

1) We don't need access to source code for detecting privacy bugs. Hence, this makes it very suitable to be employed by end-users (rather than software programmers building the system).
2) It leverages the social nature of these systems for detecting these bugs.
3) It can detect privacy bugs due to changes in the code, *i.e.,* regression testing for privacy.

There has been some recent work on data anonymization for privacy in software testing [14]–[17]. For example, when bug reports are submitted by individual users that may automatically include portions of memory, files, etc. in use by that user at the time an exception occurred. The user would like the software vendor to fix the bug, but does not want to send any identifiable personal information. In contrast, in social computing, the data is directly identified as belonging to each user and the issue is which other users can see it, not whether the software vendor can see it (of course the software vendor can see all of it!). This recent work on privacy in software testing focuses of protecting the privacy of the users by novel techniques to anonymize data; we, on the other hand, want to detect violations in privacy due to code changes or system wide policy changes. For more details on the related work, please see Section VI.

The contributions of this paper are:

- A novel software testing technique, called social testing, for the social circles of end-users to detect privacy bugs using regression testing. Social testing could potentially also be used for applications other than privacy preservation in social systems, such as in the multi-player gaming community;
- Two prototype tools that implement our technique for Facebook and Twitter; and
- A large empirical evaluation of our technique that demonstrates: (1) the feasibility and utility of our technique; and (2) the different kinds of bugs it can help detect.

II. Background and Motivation

Many recent studies on online social networks show that there is a (typically, large) discrepancy between users' intentions for what their privacy settings should be versus what they actually are [18]–[20]. For example, Madejski *et al.* report that, in their study on Facebook, 94% of their participants ($n = 65$) were sharing something they intended to hide and 85% were hiding something that they intended to share. Liu *et al.* [20] found that Facebook's users' privacy settings match their expectations only 37% of the time. This is **R1** mentioned in the previous section.

In addition to the problem of understanding existing privacy settings, there are two orthogonal problems. First, there might be software bugs in the implementation of the privacy settings, which results in over-sharing or under-sharing of information, and as software evolves over time, this might introduce new bugs. This is **R2** mentioned earlier.

Second, systems like Facebook change their policies on privacy often and these changes in policy usually end up confusing users even more. Dan Fletcher [6] writes: "In the past, when Facebook changed its privacy controls, it tended to automatically set users' preferences to maximum exposure and then put the onus on us to go in and dial them back. In December, the company set the defaults for a lot of user information so that everyone — even non-Facebook members — could see such details as status updates and lists of friends and interests. Many of us scrambled for cover, restricting who gets to see what on our profile pages." This is **R3** mentioned earlier and these are the main research problems that we are trying to solve with our approach.

III. The Social Testing Approach

The broad technique for our social testing approach is to use one's friends to help with software engineering problems. Our approach leverages the inherently social aspect of these systems, which are used for interacting and communicating with other users. This approach will apply only to systems

where users are members of possibly overlapping groups and input information intended to be shared with some of these groups they are members of but not with other groups they are members of. This includes the cases of the singleton group – just me – and the universal group – everyone who uses the system, or anyone who uses the internet since many social systems often allow certain access with no login at all.

This technique could apply towards many different functional and non-functional requirements for end-users such as privacy, performance, and so on. In this paper, we focus on privacy testing and in particular, on **R2** (detecting privacy bugs in code/API implementations) and **R3** (detecting system wide policy changes for privacy). There are two possible kinds of privacy violations:

- Over-sharing — From a user's point of view, this piece of information should have been private, but it can be viewed by others.
- Under-sharing — From a user's point of view, this piece of information should have been public, but it cannot be viewed by others.

We deal with both of these types of privacy violations. As our technique is intended for end-users, we assume that there is no access to source code. The main crux of our technique is — a user can choose his/her friends to periodically monitor what's visible to them via the social system. When they see a change in what's visible, this might be a privacy violation and they can inform the user. Thus, the aforementioned privacy violations, from the point of view of the tester, become: (1) Over-sharing — seeing more than you should; and (2) Under-sharing — seeing less than you should.

Thus, we use a social approach for detecting privacy bugs. The algorithm for our technique (from the tester's point of view) is outlined below:

1) Implement/Download/Build a wrapper that can "talk" to the system under test (via an API, screenscraping, etc.).
2) Generate a list of users to monitor.
3) Decide on the policies (how often to monitor, which things to monitor).
4) Based on the policies, use the wrapper to monitor the user(s).
5) Generate diffs (*i.e.,* differences) between the information just received and from the previous run.
6) If there is a diff, inform the user on what changed.
7) Repeat steps 4-6, as needed and update steps 2-3, when necessary.

We discuss the platform and API specific implementation issues, examples of privacy bugs that we found, and how this technique can help address **R2** and **R3** in the next section.

IV. Empirical Evaluation

For our empirical evaluation, we built two prototype tools: one for Facebook; one for Twitter. Using these tools, we evaluated how our technique could help addressing **R2** and **R3**. In particular, we had two specific research questions for our empirical evaluations:

RQ1: Feasibility — Does using our technique help in detecting privacy bugs?

RQ2: Utility — What kinds of bugs does it detect? Does it help with, both, **R2** (Code/API Bugs) and **R3** (Policy Changes)?

A. Privacy and Facebook

Facebook is a great example for doing an empirical evaluation on privacy as it follows a fine-grained privacy model. It has many different privacy parameters and options; broadly, users can choose who gets to see what type of data with a lot of granularity. It provides an API, called the Graph API, that represents the Facebook social graph using objects and connections between objects [21]. Examples of the objects include User, Events, Groups, and Applications. The User object contains fields such as name, gender, and birthday and "connections" such as albums, family, groups, likes, movies, and videos. A complete list of User Connections is available on the Facebook User API page [22].

In general, there is a lot of flexibility for the user to choose the privacy settings for all of these. Users can share the data with no one, with selected friends, with all friends, with friends of friends, with certain networks (such as "Columbia University"), and with everyone. Users can also allow certain apps to access this information.

1) Prototype Tool: Our tool is a prototype implementation of the social technique for detecting privacy bugs in Facebook. It is easily configurable and can fetch any data provided by the Facebook API. For the purposes of this study, we focused on getting only the User Connections from [22]. The prototype tool consists of two separate components: the Data Monitor and the Diff Visualizer.

The Data Monitor is implemented as a set of Ruby scripts. It uses the Koala library [23], which is a Facebook library for Ruby and supports the Graph API. The Data Monitor works as follows: First, given a list of users, for each user, it uses Koala to get data for that user. It gets all the data listed in [22] (except the "picture" connection, which didn't exist when we started collecting data). The Facebook API supports a "Batch mode" for making data requests and we use this mode to reduce the load of the servers and to get data more efficiently. Once the Data Monitor has the data, it writes it out to a log file. We create a separate log file per user. To limit the data that needs to be stored and also for privacy reasons, we do not store the entire data; we keep only the count of data items and codify the data received according to the schema defined below:

- **0, nil** — This is used when the API returns an error. This is typically a permissions error, but can also include other server side errors.
- **1, 0** — This is used when the API returns an empty data set. This means that either there is no data in that category or that the data exists but has been hidden by the user. Note that with the latter case, it will return a 1, 0 and not a 0, nil.
- **2, x** — This is used when the API returns some data. x is the count of the number of items received. For example,

```
Fri Apr 27 13:12:30 -0400 2012,
    ["someUser", [2, 259], [2, 1], [1, 0],
     [1, 0], [2, 25], [1, 0], [1, 0], [1,
    0], [2, 19], [1, 0], [2, 3], [2, 25],
    [1, 0], [1, 0], [1, 0], [2, 25], [2,
    20], [2, 25], [2, 4], [1, 0], [2, 1],
    [1, 0], [1, 0], [0, nil], [0, nil],
    [1, 0], [1, 0], [0, nil], [0, nil],
    [2, 1], [0, nil], [0, nil], [0, nil],
    [0, nil], [1, 0], [0, nil], [1, 0],
    [1, 0], [1, 0], [1, 0], [0, nil]]
Tue May 01 15:22:35 -0400 2012,
    ["someUser", [2, 259], [2, 1], [1, 0],
     [1, 0], [2, 25], [1, 0], [1, 0], [1,
    0], [2, 18], [1, 0], [2, 3], [2, 25],
    [1, 0], [1, 0], [1, 0], [2, 25], [2,
    20], [2, 25], [2, 4], [1, 0], [2, 1],
    [1, 0], [1, 0], [0, nil], [0, nil],
    [1, 0], [1, 0], [0, nil], [0, nil],
    [2, 1], [0, nil], [0, nil], [0, nil],
    [0, nil], [1, 0], [0, nil], [1, 0],
    [1, 0], [1, 0], [1, 0], [0, nil]]
```

Listing 1: Sample Log File for someUser. Note: We anonymized the username and replaced it with someUser. The order of arrays is as follows: friends, accounts, apprequests, activities, albums, books, checkins, events, feed, interests, likes, links, movies, music, notes, photos, posts, statuses, tagged, television, videos, achievements, family, friendlists, friendrequests, games, groups, home, inbox, locations, mutualfriends, notifications, outbox, payments, permissions, pokes, questions, scores, subscribedto, subscribers, updates.

if there were 10 locations that the user had been tagged at, we would log 2, 10.

The log file, thus, contains multiple lines of data. Each line contains two things: (1) The current timestamp when the data was received; and (2) An array containing the username and the 41 arrays for the connections encoded into the schema shown above. A sample log file is shown in Listing 1.

The Diff Visualizer, which is also a set of Ruby scripts, parses each log file and creates a human readable output if there is a diff (*i.e.,* difference) between any consecutive runs for a user. If there is a difference, it will print out the pairwise timestamps and what the old and the new values are. We divide a difference into two categories: a Major Difference and a Minor Difference. A Major Difference occurs when, for a certain connection, the data received changes the codifying categories. For example, if music was 2, 8 and became 1, 0, this would be a Major Difference. A Minor Difference, on the other hand, occurs when the data changes, but does not change the codifying category. For example, if music was 2, 10 and became 2, 12, this would be a Minor Difference. We, thus, have two variants of the Diff Visualizer, which will print out either Major or Minor Differences as needed. A sample output containing both Major and Minor Differences is shown in Listing 2.

```
======May 25 2012 and May 27 2012======
friends - Old: 2, 783, New: 2, 784
events - Old: 2, 1, New: 1, 0
feed - Old: 2, 15, New: 2, 16
likes - Old: 2, 202, New: 2, 200
posts - Old: 2, 6, New: 2, 7
tagged - Old: 2, 9, New: 2, 10
======May 27 2012 and May 28 2012======
friends - Old: 2, 784, New: 2, 783
posts - Old: 2, 7, New: 2, 8
tagged - Old: 2, 10, New: 2, 9
```

Listing 2: Sample Diff Output for a user. events is an example of a Major Difference; the others are all Minor Differences.

2) Feasibility: The first step in our empirical evaluation was to show the feasibility of our approach and tool. For this step, we used the tool to access the Facebook data of a research colleague of the first author. Facebook uses OAuth 2.0 [24], which is an open standard for authentication. We provided our tool with the first author's OAuth 2.0 access token so that the tool can access the same data that the first author can. This is the equivalent of the research colleague, using our social approach, asking the first author to monitor his information on Facebook. For this step of the evaluation, we did the following:

1) We accessed the Facebook data of the research colleague (name anonymized, for privacy reasons).
2) After the data was accessed, we asked the colleague to turn on privacy controls and make the data less visible. This would enable us to check if our tool could detect changes in privacy, where data is made less visible. The colleague did this by adding the first author to one of his pre-defined friend lists that had very limited access to his profile. We accessed the data using our tool again.
3) Finally, we asked the colleague to turn off the privacy controls and make the data more visible. This would enable us to check if our tool could detect changes in privacy, where data is made more visible. We accessed the data again using our tool.

The output from the Diff Visualizer is shown in Figure 1. As the figure shows, turning on the privacy settings reduces visibility — things like photos, locations, and feed were visible earlier and the Facebook API responses contained data; with the privacy settings on, the Facebook API returns an empty set. Turning off the privacy settings makes the data visible again, as seen in the right hand side of the figure.

Our Facebook prototype tool can thus detect changes in privacy settings. These changes can either be data being made more private or data being made less private. Thus, if someone suddenly starts sharing more or less data than before, our tool would detect this and this could indicate a privacy bug. Next,

```
======Apr 24 2012 and Apr 25 2012======
feed – Old: 2, 19, New: 1, 0
photos – Old: 2, 25, New: 1, 0
posts – Old: 2, 19, New: 1, 0
tagged – Old: 2, 5, New: 1, 0
videos – Old: 2, 1, New: 1, 0
locations – Old: 2, 1, New: 1, 0
```

(a) Turning On the Privacy Settings

```
======Apr 25 2012 and Apr 27 2012======
feed – Old: 1, 0, New: 2, 19
photos – Old: 1, 0, New: 2, 25
posts – Old: 1, 0, New: 2, 20
tagged – Old: 1, 0, New: 2, 4
videos – Old: 1, 0, New: 2, 1
locations – Old: 1, 0, New: 2, 1
```

(b) Turning Off the Privacy Settings

Fig. 1: Changing Privacy Settings — Output as seen from our tool

we show some examples of bugs our tool can help detect.

3) Facebook Bugs — Family and Friendlists: We ran our Facebook prototype tool using the first author's access token and collected data for all his Facebook friends ($n = 516$). The data was collected roughly every day for each user for approximately eleven weeks (from May 1, 2012 to July 20, 2012). For each user, the number of data points (*i.e.,* the number of days on which we successfully got data from the Facebook servers) was, on average, 36.24 ($\sigma = 3.26$, median = 36, max = 44, min = 25). (Please see Section V for a discussion on the number of data points and on the robustness of our approach.)

Upon running the Diff Visualizer, we found that 63.18% (326 out of 519) of the users had Major and Minor Differences during the data monitoring period. Out of these, there were a total of 5065 Minor Differences (on average, 15.54 per user) and 780 Major Differences (on average, 2.39 per user). Not all of these differences necessarily imply privacy bugs; some of these differences would arise from the "normal" use of these systems, *i.e.,* users adding new photos from a recent trip and so on. But even in these cases, the users may not be aware with whom they are now sharing this new information.

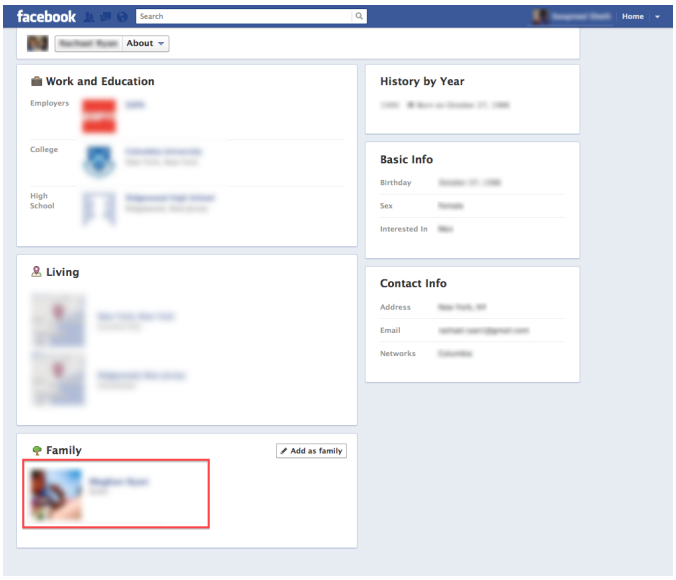
We now highlight, in this and the next subsection, a couple of interesting case studies on the bugs that were discovered using our tool. For a certain user, there was one family member being shown for the first three weeks of the data monitoring period. On May 23, 2012, our tool found a lot of Major Differences (there had been other, unrelated, Minor Differences previously) for that user. The output from our tool for the entire monitoring period is shown in Listing 3. The last diff shows a lot less data being visible. Was this a case of a user turning on privacy settings? Or perhaps did something change in the Facebook Code or API resulting in a bug? We tried to find out the cause. We were, of course, limited in our efforts as we don't have access to the source code. We decided to focus on the family connection, which lists a user's family members. This is highlighted in red in Listing 3.

The first step of our investigation was to see the actual page on Facebook. On the page, the family member was still visible. This is shown in Figure 2a, highlighted in red (user details have been blurred out to protect privacy). To ensure that there were no bugs in our tool implementation, we used the Facebook Graph API Explorer and verified that this returned an empty data set. This is shown in Figure 2b.

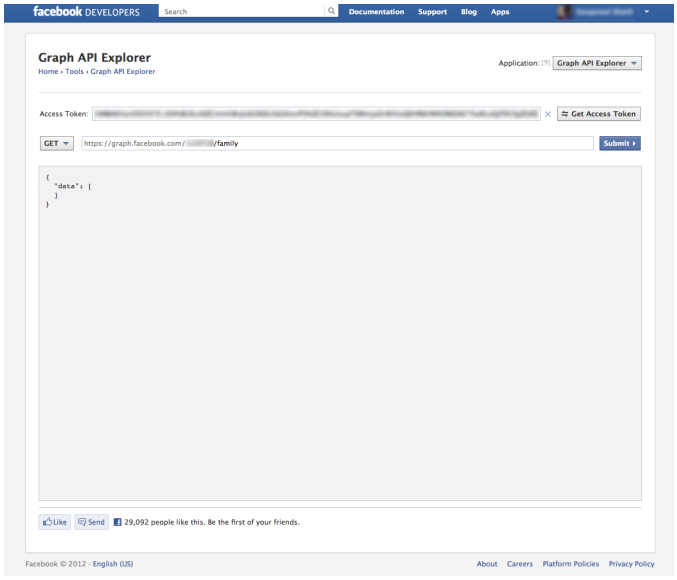
```
======May 04 2012 and May 07 2012======
feed – Old: 2, 21, New: 2, 20
posts – Old: 2, 15, New: 2, 14
======May 11 2012 and May 14 2012======
feed – Old: 2, 20, New: 2, 19
tagged – Old: 2, 11, New: 2, 10
======May 18 2012 and May 21 2012======
posts – Old: 2, 14, New: 2, 15
tagged – Old: 2, 10, New: 2, 8
======May 22 2012 and May 23 2012======
albums – Old: 2, 3, New: 1, 0
feed – Old: 2, 19, New: 1, 0
likes – Old: 2, 2, New: 1, 0
photos – Old: 2, 25, New: 1, 0
posts – Old: 2, 15, New: 1, 0
tagged – Old: 2, 8, New: 1, 0
family – Old: 2, 1, New: 1, 0
groups – Old: 2, 2, New: 1, 0
locations – Old: 2, 4, New: 1, 0
```

Listing 3: Facebook Family Bug — Output as seen from our tool

Finally, we started looking at Facebook bug reports to see if anyone else had reported a similar issue. We found two relevant bug reports. The most relevant bug report was titled "Can no longer access FriendList members on test users" [25]. In this bug report, a user had created two test users and added each user to the other's family list. When the user tried accessing the members of one of the test user's family, the Graph API returned an empty data set. This is exactly the same behavior as what we observed here. Facebook have confirmed that the bug exists and assigned it to a developer with medium priority. The second bug report, titled "Some of the friendlists do not show members from Graph API, why??" [26], reported a similar problem to the previous bug report. In this bug report, the user had many friend lists, which is a generalization of the family connection. Upon accessing the data from the Graph API, some of the friend lists return all the members; some return only a subset of the members. Facebook has responded to this bug report by triaging it with low priority.



(a) The Facebook Website — The highlighted red rectangle shows the family member.



(b) API — Using the API, the list of family members is empty.

Fig. 2: Facebook Family Bug — On the website, you can see the family member; Using the API, you cannot see the family member.

```
======Jul 02 2012 and Jul 06 2012======
feed – Old: 2, 14, New: 2, 15
links - Old: 1, 0, New: 2, 23
posts – Old: 2, 3, New: 2, 5
tagged – Old: 2, 14, New: 2, 15
```

Listing 4: Facebook Links Bug — Output as seen from our tool

These bug reports and our tool output, taken together, lead us to believe that there is a bug in the Facebook API, that was recently introduced due to code changes and should not have passed the regression testing phase. This is an example of under-sharing — less information is public than it should be. Our tool was able to detect this privacy bug using end-user regression testing.

4) Facebook Bugs — Links made public: At the start of July, our tool discovered another example of a privacy bug — this time, it was an example of over-sharing. The output from our tool for one user is shown in Listing 4. There was a lot of data made public about links posted by a user, which is highlighted in red. The interesting part was that this was not data that was recently added by the user; some of these links were added back in 2009. This was data that had been on Facebook for a while and not visible to friends of the users; now, it was visible.

Analyzing the data further, we found that 73 out of 516 users were now sharing a lot more links than before and that this new data was first visible towards the start of July. Of the 73 users, 57 (*i.e.,* 11.05%) were sharing more than one link, thus indicating that this was not new information added by the user recently.

To understand the cause behind this over-sharing, we conducted an informal open-ended interview with one of the users, Keith (name changed for privacy reasons). Excerpts from the interview are shown next (reproduced with permission from Keith). On being told that he is now sharing 23 links, which weren't visible earlier, Keith responded: "whoa [...] ok..... that is weird." After looking at the links that were visible, Keith said: "ok, all of these links are valid, but, am surprised you can see them [...] I, as a developer, opened my account for developer access. it's the only way possible and I just thought I was authorizing that one app. they must have their permissions f***ed up [...] it's either that, or facebook changed my settings automatically. " Keith said that he would change his settings back so that the links would not visible anymore and ended the interview with: "good thing your app [sic] was able to catch it."

Keith is currently a student at Columbia University pursuing a Ph.D. in Computer Science. Given that someone with a technical background didn't know about his data being public and wasn't completely sure what changed, a *non-technical end-user* would generally have a much harder time figuring out changes in privacy settings. This is the main strength of our tool — giving end-users an easy way for detecting potential privacy breaches.

Since this particular bug had not affected all the users, it seems to indicate that this a Policy Change that is being rolled out gradually by Facebook. Alternatively, this could be another example of a bug (affecting only some users) in the Facebook Code or API. Either way, our tool was able to detect this.

Field	Description
description	The user-defined UTF-8 string describing their account.
favourites_count	The number of tweets this user has favorited in the account's lifetime.
followers_count	The number of followers this account currently has.
friends_count	The number of users this account is following (AKA their "followings").
geo_enabled	When true, indicates that the user has enabled the possibility of geotagging their Tweets.
listed_count	The number of public lists that this user is a member of.
protected	When true, indicates that this user has chosen to protect their Tweets.
statuses_count	The number of tweets (including retweets) issued by the user.
verified	When true, indicates that the user has a verified account.
withheld_in_countries	When present, indicates a textual representation of the two-letter country codes this user is withheld from.
withheld_scope	When present, indicates whether the content being withheld is the "status" or a "user."

TABLE I: Partial list of Users Fields from the Twitter User API [28]

B. Privacy and Twitter

Twitter, as opposed to Facebook, has a completely different model with respect to privacy. While Facebook has a very fine-grained control model for controlling what's visible to whom, Twitter has a very coarse-grained model. Users can choose if their accounts are "protected" or not, with the account being not protected as the default setting [27]. If the account is not protected, all tweets are public and can be viewed by anyone. If the account is protected, it can only be viewed by the followers of that person. There is no mechanism for deciding this on a per-tweet basis, for example. Regardless of whether the account is protected or not, anyone can still see the number of tweets, the number of followers, and the number of following for any user.

Thus, the only setting that matters in terms of privacy is whether the account if protected or not. Our Twitter tool, described below, uses the same social approach for detecting if the privacy settings change. In addition to this, to show the generalizability of our approach in dealing with different kinds of social systems, we decided to treat some other user information as "sensitive" — *i.e.,* if Twitter had more fine-grained controls for these types of information, our approach (and tool) would still be able to detect changes in privacy settings. A partial list of user fields from the Twitter User API is shown in Table I. For the purposes of our tool and empirical study, we treated these as sensitive information as well and inform the user when these change.

1) Prototype Tool: Our prototype tool can detect privacy bugs for Twitter. Similar to the Facebook prototype tool described in Section IV-A1, it is easily configurable and can fetch any data provided by the Twitter API. For the purposes of this study, we focused on getting only the partial list of User fields shown in Table I. Similar to the Facebook tool, the Twitter tool consists of two components: the Data Monitor and the Diff Visualizer.

```
========2012-06-19 and 2012-06-19========
friends_count-Old: 2, 140, New: 2, 142
========2012-06-19 and 2012-06-19========
protected-Old: 2, false, New: 2, true
========2012-06-19 and 2012-06-19========
protected-Old: 2, true, New: 2, false
========2012-06-21 and 2012-06-22========
followers_count-Old: 2, 138, New: 2, 137
statuses_count-Old: 2, 548, New: 2, 551
```

Listing 5: Privacy Monitoring of @swapneel — Output as seen from our tool

The Data Monitor is implemented as a set of Ruby scripts. It uses the Twitter library [29] to get data from the Twitter API. The Diff Visualizer, similar the Facebook Diff Visualizer, is also a set of Ruby scripts that parses each log file and creates a human readable output if there is a diff between any consecutive runs for a user. The rest of the workings of the Data Monitor are similar to the Facebook tool described in Section IV-A1. We do not repeat the implementation details of the tools due to space limitations. The main difference is that this tool focuses on the user fields shown in Table I; the rest of the implementation is similar. The other difference is that, for Twitter due to its lack of fine-grained privacy controls, we do not distinguish between Major and Minor Differences.

2) Feasibility: We ran our Twitter prototype tool and collected data for some of the first author's research colleagues ($n = 10$). The data was collected roughly every day for each user for approximately four weeks (from May 19, 2012 to July 20, 2012).

We also collected data for the first author (@swapneel) and changed the account to protected (and back to open) and verified if the tool can detect changes in privacy settings. The tool could, indeed, pick up the changes in privacy settings. The output from the Diff Visualizer (highlighted in red) in shown in Listing 5.

One of the twitter accounts for which the data was collected was the official ICSE twitter account (@ICSEconf). If we assume that the fields listed in Table I are sensitive information, our tool can detect changes in these as well. The partial output from the Diff Visualizer is shown in Listing 6.

Recently, a bug was found in Twitter where users who wanted to follow others were "arbitrarily, randomly, and haphazardly" unfollowed [30]. This "unfollow" bug was acknowledged by the Twitter team and they said that they were working on a fix. Our tool would have been able to detect this bug as well as follows: Say I started following two new users today. If the output from the Diff Visualizer was anything other than two, we know that there is a bug with following someone. The user could then check which user got unfollowed and follow the user again, if needed.

```
========2012-06-19 and 2012-06-20========
statuses_count-Old: 2, 904, New: 2, 906
========2012-06-20 and 2012-06-21========
listed_count-Old: 2, 67, New: 2, 68
statuses_count-Old: 2, 906, New: 2, 910
========2012-06-21 and 2012-06-22========
followers_count-Old: 2, 877, New: 2, 876
statuses_count-Old: 2, 910, New: 2, 912
========2012-06-22 and 2012-06-23========
followers_count-Old: 2, 876, New: 2, 878
statuses_count-Old: 2, 912, New: 2, 913
========2012-06-23 and 2012-06-25========
followers_count-Old: 2, 878, New: 2, 879
========2012-06-25 and 2012-06-26========
followers_count-Old: 2, 879, New: 2, 880
========2012-06-26 and 2012-06-27========
followers_count-Old: 2, 880, New: 2, 882
========2012-06-27 and 2012-06-28========
followers_count-Old: 2, 882, New: 2, 883
========2012-06-28 and 2012-06-29========
followers_count-Old: 2, 883, New: 2, 885
friends_count-Old: 2, 1015, New: 2, 1016
```

Listing 6: Privacy Monitoring of @ICSEConf — Output as seen from our tool

V. Discussion

A. Flexibility

One advantage of this approach for detecting privacy bugs is the flexibility. A user could choose different sets of friends to monitor different things, if the social system has a fine-grained privacy model. For example, he could have a friend check the privacy settings of his photos and check-in locations. He could have someone from his network (such as "New York") check his music and movies. He could have a friend of a friend check his feed. He could also create overlapping groups — his friends should be able to see albums and locations; his network can only see the albums. Thus, he could use different sets of friends to verify that the privacy settings indeed are what he expects them to be and to alert him when they can see more or less than what they saw before.

A user can also choose how often the data is fetched based on how active he is on the social system, the API rate limits, and personal preferences such as the tradeoff between the load on the social system's servers and his privacy needs.

Finally, in terms of implementation, our prototype tools were stand-alone tools that ran off the command line. Social systems like Facebook and Twitter provide rich ecosystems for apps. Our approach can be implemented as apps that run on Facebook, for example. Other alternatives include a browser plugin that automatically runs the regression testing when the user logs into one of these systems or on a periodic basis (every hour, every day, and so on), a "normal" desktop application with a GUI, and so on. We used Ruby for our implementations; this was of out choice and is not a constraint. Any programming language that can access the web can be used. Having a wrapper library for that programing language does help as it obviates the need to deal with lower level protocol details. For example, Twitter has a list of libraries for 14 programming languages ranging from Java, .NET, and Python to Erlang, Scala, and Clojure [31]. There are no inherent implementation or UI limitations as far as our approach is concerned.

B. Generalizability

Another advantage of this approach for detecting privacy bugs is the generalizability of the technique. In general, it can work with any social system regardless of what kinds of privacy controls and features it has. The previous Section showed how it could work with systems at two extreme ends of the privacy spectrum: Facebook, with its fine grained settings for choosing who sees what and Twitter with its coarse grained settings, which is essentially an on/off switch.

Having an API to use makes it much easier to implement a tool for a particular social system. This, however, is not a limiting factor — if there is no API, the same approach can be combined with alternatives such as screen or web scraping.

C. Robustness

Our approach is also very robust — it does not need that the Data Monitor is run every day or that the Data Monitor successfully fetches data for each user every day (or on every run). In spite of the Facebook API having slow response times [32] and timing out occasionally, which resulted in our Data Monitor fetching data successfully on average 36 times (out of possibly about 75 times) for each user in an eleven week time period, our tool still works fine and detects bugs as shown in the previous Section. The tradeoff with fetching data less often is that we will not be able to catch transient bugs. For example, if something is made public only for a few minutes and then it is private again, if our Data Monitor is not active then, we will not be able to detect it.

In contrast to the Facebook API, the Twitter API, in our experience, was much more stable. Regardless of the stability and reliability of the social system under test, as mentioned above, our approach can still detect bugs due to its highly flexible nature.

D. Limitations and Threats to Validity

A limitation of our prototype tools is that we keep track of the number of items in data fetched, rather than the actual data. For example, in the Facebook tool, for a user, we log that the user had ten photos rather than what the photos are. We do this for two reasons: (1) to reduce the data that needs to be stored; and (2) for privacy reasons. Due to this, our tools might miss out on changes in privacy if a user, for example, deletes one photo and adds one photo, our tool would see this as no change having occurred. This, however, is a limitation of our *prototype tools*, and not of our *approach*. If an end-user wishes to keep track of the exact data, rather than the number of data items,

our tools can be modified to do that. An added challenge, in this case, would be to find the semantic similarities between data items, which would be easier in some cases (*e.g.,* checkin locations, groups) than others (*e.g.,* albums, interests, likes).

An inherent limitation of our approach is the possibility of false negatives, *i.e.,* privacy bugs that exist in the system that our tool/approach is not able to catch. There might be bugs that have existed since the first version of the software; if there is no change in the code, our regression testing approach will not work. There might be privacy bugs that affect only some of the users; if the users that are currently using our tool are not affected by this bug, we won't be able to detect it. The main reason for this is that our approach is intended for *end-users* and we don't have access to the source code of the system under test. From an *end-user* perspective, it's hard to detect bugs using our approach that may not have an external manifestation or change in behavior for our users.

Finally, coming back to our software engineering research questions, even though **R1** is beyond the scope of this paper, we make a couple of observations based on our empirical results. If there are never any changes in the social software, then **R2** and **R3** won't happen, but in a limited indirect way just trying to use our tool (which will keep saying "no change") might make users more aware of privacy settings issues and thus in a very small way help with **R1**. Now if there *are* changes, so **R2** and/or **R3** come into play, then the awareness with respect to **R1** would be stronger because users would then be prompted to go look more closely at the particular settings that were affected and thus would understand them better and adjust their mental model accordingly.

Statistical Conclusion — Do we have sufficient data to make our claims? For our Facebook tool, we fetched data for 516 users resulting in almost 18,700 data points. For our Twitter tool, we fetched data for 10 users resulting in almost 350 data points. The goal of the empirical evaluations was to find examples of privacy bugs to show the feasibility and utility of our approach, which we did find as described in Section IV.

External Validity — Do our results generalize to other systems? Our prototype tools were implemented for two different systems: Facebook and Twitter. Our approach is broad and can apply to any social system as discussed in Sections V-A and V-B above.

VI. Related Work

There have been some recent papers on data privacy and software testing. Clause and Orso [14] propose techniques for the automated anonymization of field data for software testing. They extend the work done by Castro *et al.* [33] using novel concepts of path condition relaxation and breakable input conditions resulting in improving the effectiveness of input anonymization. Taneja *et al.* [15] and Grechanik *et al.* [16] propose using k-anonymity [34] for privacy by selectively anonymizing certain attributes of a database for software testing. These papers propose novel approaches using static analysis for selecting which attributes to anonymize so that test coverage remains high. Peters and Menzies [17] propose an anonymization technique for sensitive data so that it can be used for cross-company defect prediction. They show that it is possible to make data less sensitive and still maintain high utility for data mining applications.

Our work is orthogonal to these papers on data anonymization. The problem they address is — how can one anonymize sensitive information before sharing it with others (*e.g.,* sending it to the teams or companies that build the software, sharing information for testing purposes, and sharing data across multiple companies, respectively)? The problem we address is - how can *end-users* verify if the software systems they are using are handling privacy correctly?

Further, all these papers are trying to protect the privacy of the data. We, on the other hand, are trying to detect privacy violations and test if the systems have any privacy bugs.

There has been a lot of work in the field of regression testing mainly towards test case selection and test case prioritization [35]–[38], including a very detailed, and excellent, recent survey by Yoo and Harman [39] and the references therein. Our work builds on, and differs from, all of the above in two aspects – our regression testing approach is targeted towards *end-users* and is targeted towards finding *privacy* bugs.

There has also been some recent work in using taint analysis for detecting security and privacy violations [40], [41]. These approaches require access to source code for taint analysis. Our approach, on the other hand, is targeted towards end-users who do not have source code access to the social systems that they are using.

Our social testing approach is similar in some ways to "do you see what I see," a technique proposed in the networking community to support distributed fault detection and diagnosis from the client-side [42], although there the actual end-users are not directly involved, and is also related to the network security communitys collaborative intrusion detection, e.g., [43], where the goal is to share data about penetration attempts against different organizations' enterprise networks but without inadvertently sharing any private information.

VII. Conclusion

Privacy in social systems is becoming a major concern. End-users of such systems are finding it increasingly harder to understand the complex privacy settings. Even if they do understand the settings, as the software evolves over time, there might be bugs introduced that breach users' privacy. There might also be system wide policy changes that could change users' settings to be more or less private than before.

We present a novel technique, called ***Social Testing***, that can be used by *end-users*, as opposed to software developers building the system, for detecting changes in privacy, *i.e.,* regression testing for privacy. This technique can broadly apply towards functional and non-functional requirements for end-users such as privacy, performance, and so on. In this paper, we applied our technique towards detecting privacy bugs from an end-user perspective. Broadly, a user can use his/her friends to monitor what information is visible in the social systems

and to automatically detect when more or less information is visible, thus indicating a potential privacy concern. We also presented two prototype tools — one for Facebook ; one for Twitter — that implemented our technique for detecting privacy bugs. The results of our evaluation show the utility and feasibility of our approach and tools for detecting privacy bugs. Our Facebook tool discovered that 63.18% of the users had differences in privacy where they were sharing either more or less information than before.

In particular, we focused on two case studies of bugs that we found and upon interviewing one user affected by the bug, the user said: '[. . .] am surprised you can see them (new information that was recently made visible, which was detected by our tool) [. . .] good thing your app was able to catch it" and that he would change his settings back to what they should have been. To the best of our knowledge, this is the first technique that leverages regression testing for detecting privacy bugs from an end-user perspective.

ACKNOWLEDGMENT

The authors are members of the Programming Systems Laboratory funded in part by NSF CCF-1161079, NSF CNS-0905246, and NIH 2 U54 CA121852-06.

REFERENCES

[1] M. Barbaro, T. Zeller, and S. Hansell, "A face is exposed for AOL searcher no. 4417749," *New York Times*, vol. 9, 2006. [Online]. Available: http://www.nytimes.com/2006/08/09/technology/09aol.html?_r=1&pagewanted=all

[2] M. Shiels, "Germany officials launch legal action against Facebook," http://news.bbc.co.uk/2/hi/technology/8798906.stm, July 2010.

[3] R. Richmond, "Gadgetwise: A Guide to Facebook's New Privacy Settings," http://gadgetwise.blogs.nytimes.com/2010/05/27/5-steps-to-reset-your-facebook-privacy-settings/, May 2010.

[4] L. Rich, "A guide to protecting your privacy on Facebook," http://news.bbc.co.uk/2/hi/programmes/click_online/8717750.stm, June 2010.

[5] B. Bosker, "Facebook CEO 'Doesn't Believe In Privacy'," http://www.huffingtonpost.com/2010/04/29/zuckerberg-privacy-stance_n_556679.html, April 2010.

[6] D. Fletcher, "How Facebook Is Redefining Privacy," http://www.time.com/time/business/article/0,8599,1990582.html, May 2010.

[7] S. Johnson, "Web Privacy: In Praise of Oversharing," http://www.time.com/time/business/article/0,8599,1990586.html, May 2010.

[8] M. Zuckerberg, "Making Control Simple," http://blog.facebook.com/blog.php?post=391922327130, May 2010.

[9] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011.

[10] Facebook, "Facebook Developers," http://developers.facebook.com/, 2007.

[11] Twitter, "Twitter Developers," https://dev.twitter.com/, 2008.

[12] Last.fm, "API," http://www.last.fm/api, 2009.

[13] Google Developers, "Google+ API – Google+ Platform," https://developers.google.com/+/api/, 2011.

[14] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11, 2011, pp. 21–30.

[15] K. Taneja, M. Grechanik, R. Ghani, and T. Xie, "Testing software in age of data privacy: a balancing act," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. SIGSOFT/FSE '11, 2011, pp. 201–211.

[16] M. Grechanik, C. Csallner, C. Fu, and Q. Xie, "Is data privacy always good for software testing?" *Software Reliability Engineering, International Symposium on*, vol. 0, pp. 368–377, 2010.

[17] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with morph," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 189–199.

[18] M. Madejski, M. Johnson, and S. M. Bellovin, "A study of privacy settings errors in an online social network," *Pervasive Computing and Comm. Workshops, IEEE Intl. Conf. on*, vol. 0, pp. 340–345, 2012.

[19] M. Johnson, S. Egelman, and S. Bellovin, "Facebook and privacy: Its complicated," in *Symp. on Usable Privacy and Security*, 2012.

[20] Y. Liu, K. P. Gummadi, B. Krishnamurthy, and A. Mislove, "Analyzing facebook privacy settings: user expectations vs. reality," in *Proc. of the 2011 SIGCOMM Conf. on Internet measurement conf.*, 2011, pp. 61–70.

[21] Facebook, "Graph API," https://developers.facebook.com/docs/reference/api/, 2007.

[22] ——, "User," https://developers.facebook.com/docs/reference/api/user/, 2007.

[23] Alex Koppel, "Koala," https://github.com/arsduo/koala/, April 2010.

[24] E. Hammer-Lahav, D. Recordon, and D. Hardt, "The OAuth 2.0 authorization protocol," http://tools.ietf.org/html/draft-ietf-oauth-v2/, 2010.

[25] Facebook Developers, "Bugs – Can no longer access FriendList members on test users," https://developers.facebook.com/bugs/368623589859564, June 2012.

[26] ——, "Bugs – Some of the friendlists do not show members from Graph API, why??" https://developers.facebook.com/bugs/400876833291706, April 2012.

[27] Twitter, "About Public and Protected Tweets," http://support.twitter.com/entries/14016, 2012.

[28] Twitter Developers, "Users," https://dev.twitter.com/docs/platform-objects/users, 2012.

[29] J. Nunemaker, W. Netherland, E. Michaels-Ober, and S. Richert, "The Twitter Ruby Gem," http://twitter.rubyforge.org/, 2006.

[30] J. Owyang, "Coping With Twitters Unfollow Bug," http://techcrunch.com/2012/03/27/unfollowbug/, March 2012.

[31] Twitter Developers, "Twitter Libraries," https://dev.twitter.com/docs/twitter-libraries/, 2012.

[32] S. Perez, "Facebook Wins "Worst API" in Developer Survey," http://techcrunch.com/2011/08/11/facebook-wins-worst-api-in-developer-survey/, August 2011.

[33] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Proc. of the 13th Intl. Cong. on Arch. support for Prog. languages and operating systems*, 2008, pp. 319–328.

[34] L. Sweeney, "k-anonymity: a model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 5, pp. 557–570, 2002.

[35] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proc. of the 2008 Intl. Symp. on Software testing and analysis*, 2008, pp. 75–86.

[36] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proc. of the 2009 Intl. Symp. on Software testing and analysis*, 2009, pp. 213–224.

[37] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 133–143.

[38] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," in *Proc. of the 16th ACM SIGPLAN Conf. on OO prog., systems, languages, and appl.*, 2001, pp. 312–326.

[39] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[40] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of the 9th USENIX Conf. on Operating systems design and impl.*, 2010, pp. 1–6.

[41] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 317–331.

[42] V. Singh, H. Schulzrinne, and K. Miao, "Dyswis: An architecture for automated diagnosis of networks," in *Network Operations and Management Symposium*. IEEE, 2008, pp. 851–854.

[43] J. J. Parekh, K. Wang, and S. J. Stolfo, "Privacy-preserving payload-based correlation for accurate malicious traffic detection," in *Proc. of the SIGCOMM Workshop on Large-scale attack defense*, 2006, pp. 99–106.