Hierarchical Data-Flow Graphs

José Pereira
Checkmarx, Braga, Portugal

Vitor Vieira
Checkmarx, Braga, Portugal

Alberto Simões
Checkmarx, Braga, Portugal
2Ai, School of Technology, IPCA, Barcelos, Portugal

Abstract

Data-Flows are crucial to detect the dependency of statements and expressions in a programming language program. In the context of Static Application Security Testing (SAST), they are heavily used in different aspects, from detecting tainted data to understanding code dependency.

In Checkmarx, these data flows are currently computed on the fly, but their efficiency is not the desired, especially when dealing with large projects. With this in mind, a new caching mechanism is being developed, based on hierarchical graphs.

In this document, we discuss the basic idea behind this approach, the challenges found and the decisions put in place for the implementation. We will also share the first insights on speed improvements for a proof of concept implementation.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Software and its engineering → Compilers; Theory of computation → Graph algorithms analysis

Keywords and phrases Data Flow, Static Application Security Testing, Hierarchical Graphs

Digital Object Identifier 10.4230/OASIcs.SLATE.2023.11

Funding *Alberto Simões*: This paper was partially funded by national funds, through the FCT/MCTES of the projects UIDB/05549/2020 and UIDP/05549/2020.

1 Introduction

SAST (Static Application Security Testing) [8] is one of the different techniques employed by Checkmarx for analyzing source code and scanning it for security vulnerabilities. As the name implies, SAST tools scan the source code without executing it. The identification of potential security weaknesses is performed after constructing an abstract syntax tree (AST) for the code being analyzed and using a query system to find specific code patterns. SAST can be used to detect vulnerabilities such as SQL injection, cross-site scripting, or buffer overflow situations.

One important feature of SAST tools is the possibility to compute data flows. Data flows allow the understanding of which expressions have their values affected by the values of variable declarations or other expressions. This is useful, as an example, to understand if a query to a database might be influenced directly by the user input, or if, during the data flow, there is any kind of sanitization[1] preventing SQL Injection. One of the first works using data flows to analyze software reliability was conducted by Fosdick and Osterweil (1974) [4]. Their work includes a comprehensive explanation of what are data flows and how to represent them as a graph. Data flows are an important part of SAST implementations [7].

[1] Sanitization is the term used to any code that prevents the vulnerability to occur, remediating it.

The Checkmarx SAST engine does data flow analysis on demand, meaning that, whenever a calculation is requested, rather than traversing a graph looking for paths between sources and sinks[2], it computes which are the next AST nodes to visit checking what immediate data is affected by the first. This process iterates until a destination is eventually reached, or there are no more adjacent nodes.

This might seem inefficient but, in fact, the traditional strategy of path-finding on an actual graph was used before and it produced results 30% slower on average than the current solution. The cause for this is the unnecessary graph expansion for every node on the abstract syntax tree, even when no flow calculation is requested from them, allied with the fact that the computational weight of finding a path in a large graph is heavier than computing adjacency between nodes. Still, the process consumes a considerable portion of scan time, averaging 30% scan time for most source projects.

The proposal is to cache a graph in which there's enough context to avoid searching impossible paths when specific sources and sinks are used. For that, clusters of vertices are created, each one representing an entry or exit point of a certain context. The idea is to be able to match any node on the AST to a cluster of vertices and therefore understand if it is worth exploring for a specific path. There can be several kinds of grouping strategies for the clusters as we will discuss later. For the Proof-of-Concept (POC) described here, nodes were grouped functions/methods in which they are encapsulated.

In the next Section, a small literature review is presented focusing on the main concepts used in the implementation of this solution. Follows Section 3 where the current data flow engine algorithm is explained, allowing a better understanding of the proposed approach. Section 4 describes the POC, including a first evaluation of the obtained results. Section 5 concludes with some final remarks and describes future work.

2 Literature Review

As a first note, Data Flows and Static Application Security Testing are not new concepts. As shown in this section, most of the concepts have more than 50 years. Nevertheless, these concepts are the base that support the development of the Proof-of-Concept here described.

In 1976, Allen and Cocke [1] present multiple situations in which a program data flow can be of use. One of the most relevant for this work is to know what data use might be affected by a particular variable definition, and the inverse, for a given use of a variable, the definitions which can potentially supply values to it. They provide a formal definition of what a data flow is, which might be summarized as a connected, directed graph with a single entry point (usually a variable declaration), and where each vertex[3] represents a statement or expression whose value depends on the graph entry point or an expression or part of an expression which modifies that data item.

As referred to in the introduction, Fosdick and Osterweil [4] present in their paper the definition of a data flow, including a couple of examples of how to represent them as directed graphs, as well as how these flows can be used to detect software implementation problems. They present a system to analyze Fortran code, called DAVE, that detects some of the most common data flow anomalies. DAVE performs this analysis by computing a flow graph search for each variable in a given unit and analyzing subprograms. This is, probably, one of the first SAST implementations.

[2] The term *source* or *input* are traditionally used as the first node in the flow search, while the *sink* is the target node (or target nodes).

[3] For clearness, the term *node* will be used to refer to an item in the Abstract Syntax Tree, and the term *vertex* will be used to refer to graph vertices. In most cases, a vertex represents a node, and therefore, some confusion may arise.

In their book [6], Khedker, *et al.* describe many different approaches for data flow analysis, and how they can be used to find different situations. The book finishes with a chapter on implementing data flow analysis in C programs using the GCC C compiler.

There are other examples of data flow analysis, that are not listed here, as they are focused on a single programming language, as the examples shown above.

3 Checkmarx Lazy Flow

Checkmarx SAST solution supports data flow computation. It is executed on demand, every time a specific flow is required. The flow can be computed in the flow direction (execution flow) or backwards. This allows the engine to choose the direction that promises a relatively smaller number of paths. This feature is known as Lazy Flow and is used by most queries used to find vulnerabilities.

The Lazy Flow process receives a set of input nodes from the Domain Object Model[4] (DOM), and a set of sink nodes. The algorithm is also able to deal with a set of sanitizer nodes. These nodes are user-defined and consist of a set of specific instructions that should be considered a flow barrier. As an example, consider the storage of personal information as a password in a database. The flow could be discarded if the password gets encrypted (thus making the flow not vulnerable). The encryption functions act as barriers and are considered sanitizers.

The Lazy Flow algorithm considers each expression or statement as a potential hop in the flow. Each hop has two visitors that decide the possible next (or previous) nodes to look up, according to the flow direction. The search ends whenever a sanitiser node is found or when a maximum number of hops was visited. The algorithm only considers the shorter path between two specific nodes for efficiency.

This flow computation is performed on-the-fly, every time a flow is requested as shortly described in the Section 1. This means that highly reused code blocks are scanned over and over again for different vulnerability detection, producing a time overhead. This is the main problem the proposed POC tries to tackle.

Note that the fact that the graph is not persistent (there is not a proper graph representation for the possible flows) does not have a real impact in terms of performance, as the Domain Object Model acts as the graph, and only in very specific situations the edge computation is not immediate.

4 Data Flow Hypergraph

A common solution for reducing the complexity of a graph is creating an abstraction that clusters vertices together. Each cluster becomes a new vertex, and edges between these clusters are aggregations of the original edges. Inside each cluster, a vertex is a graph.

These data structures are usually referred to as hierarchical graphs and are well-studied. The path-finding algorithms perform at the top level and, when a concrete path is required, look inside the relevant clusters to compute the real path.

These structures are heavily used in navigation, being in artificial or real worlds, and therefore applications are found in the areas of video games, robotics and geographic information systems. Examples of applications of Hierarchical Graphs (HG) include Pelechano

[4] The Domain Object Model can be perceived as an Abstract Syntax Tree whose structure is shared among different languages, and that does not mimic exactly the parsed code, but its semantics.

and Fuentes (2016) [9] work for path-finding in Meshes, or the work by Antikainen (2013) [2] use of HG for non-uniform traversal costs. Other examples can be found in the literature [5, 11, 3]. We will use the term Hypergraph to refer to the concrete HG implementation.

For the use case under consideration, one of the first discussions is about the clustering approach. How to consider two nodes from the DOM to be part of the same cluster? This discussion will be presented in Section 4.1. Follows Section 4.2 with an in-depth explanation of the adopted clustering approach. Section 4.3 explains how the HG is being used and presents some analysis of the obtained results.

4.1 Clustering DOM Nodes

One first decision to take is how to cluster the DOM nodes, to produce the hierarchical graph. The approach to cluster the nodes can be defined accordingly with different semantic approaches. The two main ideas discussed were:

- Cluster nodes by the file in which they appear. This was the first idea given the parallel work on an incremental parsing mechanism that deals with the change of a single file in a project repository. This clustering would make the process of updating the HyperGraph easy. Nevertheless, there is no clear definition of what a flow inside a file is. While in some languages or some projects that might exist, a simple file that is just a library would be hard to be properly grouped as a cluster, as it would have an extremely large number of inbound and outbound edges.
- Cluster nodes by the function in which they appear. This would mean that a vertex in the Hypergraph would be a function and connections will represent the flows that enter or exits that function. While the number of vertexes will rise, compared with the previous approach, there is a clear semantic meaning of edges: they are method invocations or stack frames. Curiously, this is quite similar to the second approach proposed by Sharir and Pnueli [10] in 1981.

While the chosen approach was to cluster based on functions, and as it will be seen in the next sections, some changes on the original idea were performed to encompass different entry and exit points from methods. This will be described in the next section.

4.2 Method-based Clustering

For this POC, the chosen strategy to cluster vertices was by method/function invocation. The term method will be used to refer to both functions and methods because the DOM was designed for object-oriented programming languages and despite being able to also support other paradigms, functions are wrapped in default classes for the sake of compatibility, making them static methods.

A DOM node belongs to a specific method cluster if it is under the methods sub-tree. Therefore, computing the first ancestor which is a method declaration is enough to infer the vertex the node belongs to. Vertices on the Hypergraph should be entry points of methods, such as parameters, and exit points, like return statements. Note that data can flow in and out of methods through other kinds of nodes. Arguments can lead us to other clusters/methods and method calls can make data flow into a method.

Edges on the Hypergraph are flows between the entry and exit points. Whenever there are nodes in between this flow, these sequences are stored in the graph, annotating it. This sequence represents the data flow from the method entry point to an exit point.

Follows a simple example. Consider the C#-like code sample in Listing 1, which describes a basic program that would take in an input string and execute one SQL command, that is affected by that same input.

Listing 1 Sample C# code with interprocedural calls.

```
void main() {
    string someInput = readFromStdIn();
    handleInput(someInput);
}

void handleInput(string someInput) {
    if(isBadInput(someInput)) {
        printErrorMessage(someInput);
        abort();
    }
    else {
        string preparedStatement = prepareStatement(someInput);
        executeSqlStatement(preparedStatement);
    }
}

string prepareStatement(someInput) {
    return "SELECT␣*␣FROM␣USERS␣WHERE␣USERNAME␣=␣" + someInput;
}

void executeStatement(string sqlStatement) {
    printResults(database.Execute(sqlStatement));
}
```

Figure 1 shows that same code, where some nodes are highlighted. In green, we have entry data points, and in red we have exit points. Entry points are, mostly, parameter declarations (lines 7, 21 and 26) and method calls (lines 3, 9, 16 and 28). Exit points are return statements (line 23) or parameters inside method calls (lines 4, 9, 11, 16, 17 and 28).

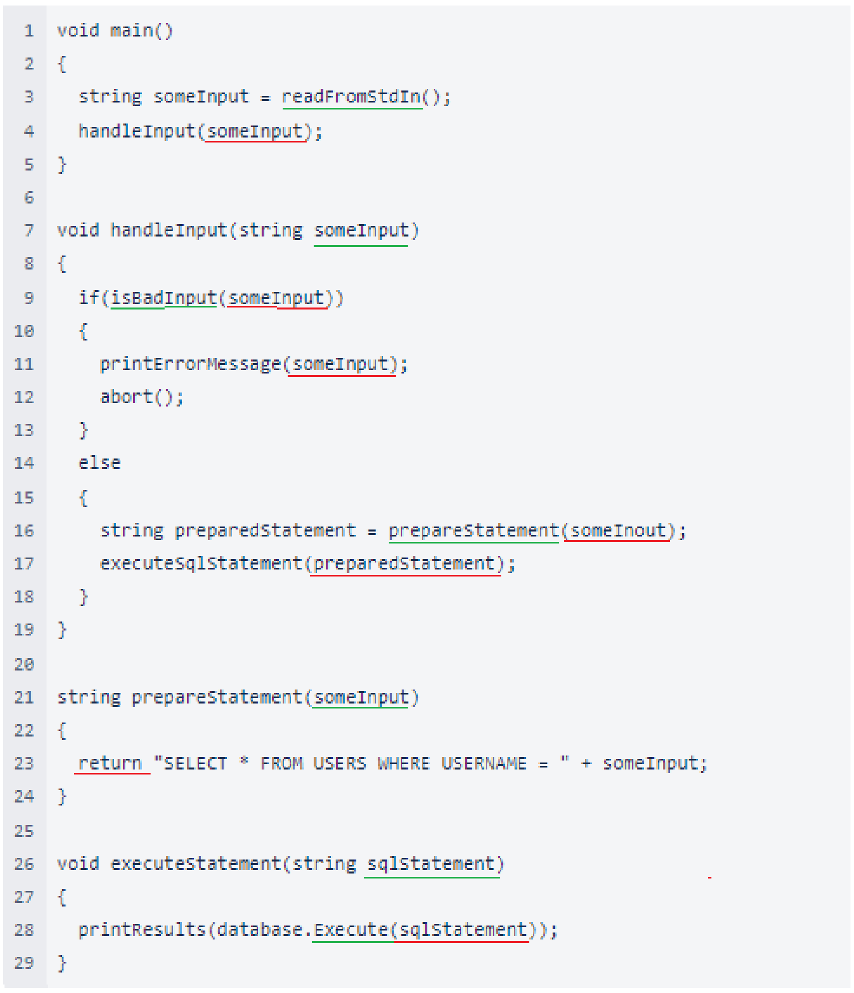
These will be the vertices on the Hypergraph. For the edges, we can compute the data flow between these nodes and aggregate the sequences between entry and exit points of the same method. For simplicity, Figure 2 shows only the relevant flows between the entry and exit nodes. In this image, the purple arrows are edges between an entry point and an exit point of a method, while the yellow arrows are edges between clusters.

The graph is stored in QuickGraph[5] library using specific vertex and edge definitions. Vertices include the entry or exit node information, and edges include the full path between these nodes. Looking at Figure 2, purple arrows include paths, while yellow arrows are just empty edges. The graph is bidirectional, thus allowing the computation of forward and backward flows.

4.3 Hypergraph Application and metrics

While the final implementation will feature a rewritten flow engine, that will take advantage of the Hypergraph to decide which paths are worth exploring, following the usual implementation of hierarchical graphs, currently the POC uses the Hypergraph information as a cache, fast-forwarding the computation of flows inside methods.

[5] `https://kernelith.github.io/QuikGraph/`



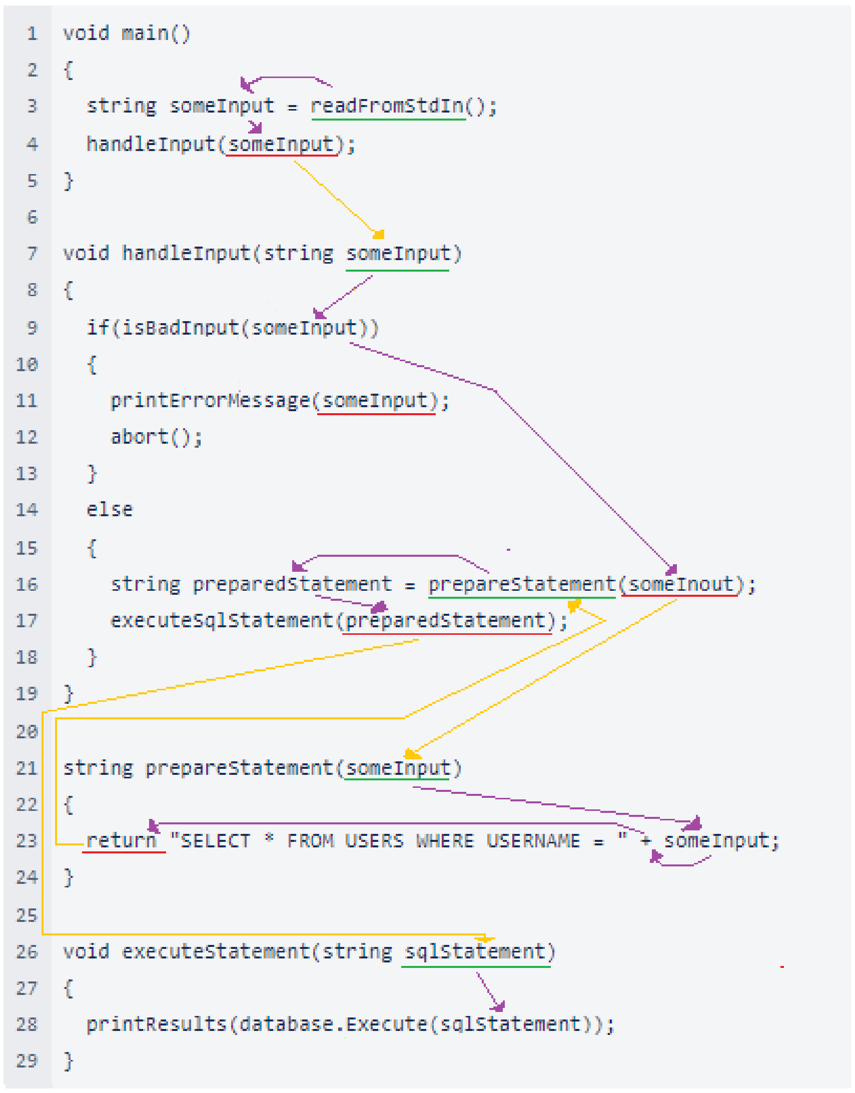
```
void main()
{
  string someInput = readFromStdIn();
  handleInput(someInput);
}

void handleInput(string someInput)
{
  if(isBadInput(someInput))
  {
    printErrorMessage(someInput);
    abort();
  }
  else
  {
    string preparedStatement = prepareStatement(someInout);
    executeSqlStatement(preparedStatement);
  }
}

string prepareStatement(someInput)
{
  return "SELECT * FROM USERS WHERE USERNAME = " + someInput;
}

void executeStatement(string sqlStatement)
{
  printResults(database.Execute(sqlStatement));
}
```

Figure 1 Inbound (green underlines) and outbound flow nodes (red underlines).

Consider an input node and a sink. Given the input node, and considering it is not in the current function scope, the traditional flow algorithm is computed until a relevant node is visited: a method call, a parameter declaration, a return statement, or a parameter in a method call. These are the entry and exit points for the Hypergraph, as stated earlier. At this point, the Hypergraph is queried. If the sink node is inside the Hypergraph cluster of nodes, the traditional flow algorithm keeps in charge. If not, the Hypergraph cluster node is used and the path is fast-forwarded until the end of the flow (next flow jump). While this process does not reduce the amount of visited paths it reduces the amount of calls to the path-finding algorithm. This would result in efficiency improvements for large methods, and little or even an efficiency decrease for small methods.

This prototype implementation was used to measure the performance impact on data flow calculations of some benchmark projects[6], producing the results presented in table 1.

[6] These are some open-source projects written in different languages, that are used internally for benchmark purposes.



```
void main()
{
  string someInput = readFromStdIn();
  handleInput(someInput);
}

void handleInput(string someInput)
{
  if(isBadInput(someInput))
  {
    printErrorMessage(someInput);
    abort();
  }
  else
  {
    string preparedStatement = prepareStatement(someInout);
    executeSqlStatement(preparedStatement);
  }
}

string prepareStatement(someInput)
{
  return "SELECT * FROM USERS WHERE USERNAME = " + someInput;
}

void executeStatement(string sqlStatement)
{
  printResults(database.Execute(sqlStatement));
}
```

Figure 2 Flows inside functions: purple arrows are paths along lines of code while yellow arrows are just empty edges.

The first project was the only one with a positive impact from the altered Lazy Flow algorithm, using the Hypergraph. By correlating flow calculation statistics between these projects, several things can be observed. AccorStruts is the project leading in terms of time spent searching for the next references from the total time of flow calculation. Reference finding is the act of mimicking the program's control flow in order to understand which symbol reference is the data flowing into next. This is a costly procedure that has specific logic for each different DOM node.

One other observation, looking into the paths returned by the Lazy Flow algorithm, is that the number of resulting flows is different when using the unaltered Lazy Flow and the version using the Hyperhraph. This is an indicator that the algorithm is probably lacking context and making some wrong assumptions about node sequences when compared to the pure Lazy Flow approach. Finally, the only coherent and significant number regarding the number of path reuses from the Hypergraph is with the AccorStruts project. Every other

Table 1 Times before and after the Fast-Forward implementation (LOC=Lines of Code).

Project	LOC	Query time	Query time with FF
AccorStruts	79.857	02:32.3	01:35.7
WebGoat	117.234	10:39.4	11:09.6
Qmxpp	20.478	00:24.0	00:30.3
Bookstore	17.588	00:25.9	00:43.0

project has a very low number of reuse, meaning that either the methods are used only once per flow, that the clustering approach was not the best, or simply implies bugs in the implementation.

5 Conclusions and Future Work

In this article, we present a first approach to develop a proof of concept for a hierarchical graph to compute data flows for SAST. We are still in the early stage of the process, with a prototype that is already allowing the analysis of the graph reuse and its impact on the flow computation efficiency. Nevertheless, a lot of effort is still required to have a fully operational solution. And more investment should be put into the POC to extract clearer conclusions.

In the making of this article, the generation of the Hypergraph and its reuse was achieved by piggybacking the LazyFlow logic. The next steps would be to match the exact same results as the standard Lazy Flow with the HyperGraph approach, creating a proper base for a benchmark. After, the actual idea that served as motivation for the POC should be put into practice. For that, a custom graph path-finding algorithm needs to be developed, taking into account the same context that Lazy Flow uses. For instance, one cannot leave a method cluster through means of a return statement into a different instance from where it was entered.

Another point that requires further improvement is the use of output or reference parameters, that are available in some languages. At this point, those were deliberately ignored to have a simpler setup for initial analysis of the improvements resulting from this approach.

References

1 F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, March 1976. `doi:10.1145/360018.360025`.

2 Harri Antikainen. Using the hierarchical pathfinding a∗ algorithm in GIS to find paths through rasters with nonuniform traversal cost. *ISPRS International Journal of Geo-Information*, 2(4):996–1014, October 2013. `doi:10.3390/ijgi2040996`.

3 Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):1–22, 2004.

4 Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976. `doi:10.1145/356674.356676`.

5 Matthias Grundmann, Vivek Kwatra, Mei Han, and Irfan Essa. Efficient hierarchical graph-based video segmentation. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2141–2148, 2010. `doi:10.1109/CVPR.2010.5539893`.

6 Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, March 2009.

7 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Data flow analysis. In *Principles of Program Analysis*, pages 35–139. Springer Berlin Heidelberg, 1999. `doi:10.1007/978-3-662-03811-6_2`.

8 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1999. `doi:10.1007/978-3-662-03811-6`.

9 Nuria Pelechano and Carlos Fuentes. Hierarchical path-finding for navigation meshes (HNA$*$). *Computers & Graphics*, 59:68–78, October 2016. `doi:10.1016/j.cag.2016.05.023`.

10 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Programme Flow Analysis*, pages 189–233. Prentice Hall, April 1981.

11 Edgar-Philipp Stoffel, Korbinian Schoder, and Hans Jürgen Ohlbach. Applying hierarchical graphs to pedestrian indoor navigation. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '08, New York, NY, USA, 2008. Association for Computing Machinery. `doi:10.1145/1463434.1463499`.