Exploiting the Structure of Modern Web Applications

Aaron Michael Blankstein

A Dissertation
Presented to the Faculty of Princeton University in Candidacy for the Degree of Doctor of Philosophy

Recommended for Acceptance by the Department of Computer Science Adviser: Michael J. Freedman

September 2017
Abstract

In this thesis, I show how, in many respects, modern web applications are built with implicit structure, such that several classic problems in computer systems can be tackled in new ways, allowing today’s applications to readily reap security or performance benefits. I apply this to two separate contexts. In the first, I look at the problems of providing some automatic security to web applications by partitioning the server-side code of the application, isolating those partitions, learning access control policies for those partitions, and extending this isolation to front-end code with JavaScript sandboxing. In the second, I look at how modern web applications interact with application caches, and develop a family of cache eviction policies tailored to these needs by focusing on prioritizing individual items to capture how item request frequencies, associated costs, and expiration times affect cache performance.
Acknowledgements

I would like to start by thanking my adviser, Mike Freedman, for the advice and feedback you provided throughout my graduate career. This dissertation has been a long and winding road, and without your guidance, I am positive that I would have gotten lost along the way. You ensured that I focused on the interesting problems in my projects and helped steer me away from the pitfalls which I would have otherwise found so enticing. You made sure that I approached opportunities throughout my graduate career with a clarity and focus, and for that, I will always be indebted. I would also like to thank Siddhartha Sen, whose assistance and contributions to the project comprising the third chapter of this thesis are very appreciated. That summer internship at Microsoft Research contributed greatly to the project and enabled me to interact with a diverse range of researchers. Your advice on research and life has been a continuous gift.

I'd like to thank the students and faculty at Princeton who made the department such a vibrant place for me to work. To the former, current, and future members of the SNS lab, who provided much needed feedback on my work and presentations, thank you. I feel particularly indebted to those students who included me in their projects over the years. Ari Feldman and Marcela Melara, assisting on your projects was a genuine privilege. Thank you also to Ed Felten, whose conversations and advice on my projects have always been appreciated. I must also thank my thesis committee for providing feedback that helped to ensure the ultimate coherence of this thesis.

To my friends who helped make my graduate school experience a joy, thank you. Thank you to Alex and Jono, who so generously taught me how to climb. To my roommates over the years, Ed, Pawel, Jordan, Tanya, Max, and Maia, thank you for putting up with me. I know it wasn’t easy. Thank you also to my family. To my Aunt Beth, Uncle Ari, Eli and Noah, thank you for ensuring that I always felt welcome in this northern reach of I-95. Thank you to my cousin Asher for providing
me with my first computer textbook, and included compiler, on my 11th birthday. To my parents, thank you for your unconditional love and support over all these years. Thank you for placing me in front of a computer and ensuring that I inherited the spirit of a tinkerer. Computers have really been the best toys. I must also thank my sister, Sarah, who has always been the best sibling I could have hoped for and a perfect role model for me to imitate. Finally, Shayna, thank you for everything, but most immediately thank you for seeing me through the process of writing this dissertation and ensuring that I would actually complete it.

Support. The research in this thesis was supported by the following sources. The research comprising Chapter 2 of this thesis was supported by NSF Award CSR-0953197 (CAREER) and a Sloan Research Fellowship. The research comprising Chapter 3 of this thesis was supported by the same NSF Award CSR-0953197, and part of the work was conducted during an internship at MSR NYC.
To my parents.
# Contents

Abstract ................................................................. iii
Acknowledgements ......................................................... iv
List of Tables .......................................................... xi
List of Figures ........................................................... xii

1 Introduction ......................................................... 1

1.1 Structure of Modern Web Applications ......................... 3
  1.1.1 Separating State from Application Logic .................... 4
  1.1.2 Easing Data Separation with Web Frameworks .......... 5

1.2 Usage and Behavior of Modern Web Applications .......... 6

2 Protecting Web Applications with Passee ...................... 8

2.1 Passee Overview ................................................... 8

2.2 Security Goals and Assumptions ................................ 13
  2.2.1 Threat Model .................................................. 13
  2.2.2 Motivating Classes of Vulnerabilities .................... 14
  2.2.3 Security Properties of Passee ............................... 15

2.3 Passee Design ...................................................... 16
  2.3.1 Interacting with a Shared Data Store .................... 18
  2.3.2 Protecting the Shared Data Store ......................... 19
  2.3.3 Learning Constraints ....................................... 20
List of Tables

2.1 Captured application policies and code coverage. . . . . . . . . . . . . 42
2.2 Number of queries per response in benchmarked applications. . . . . 50
3.1 Miss rates on Decision Service and Viral Search apps. . . . . . . . . 80
3.2 Performance improvement of django-wiki with Hyperbolic Caching. 81
3.3 NodeJS performance with Hyperbolic Caching. . . . . . . . . . . . . . 86
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Multi-tiered App Architecture</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>MVC as interpreted by Django web framework</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>A typical tiered service architecture.</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>High-level overview of the Passe runtime</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Queries demonstrating dependency types.</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Communication overview in Passe runtime</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Cryptographic token passing in Passe.</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Cross-view scripting attacks.</td>
<td>27</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of allowed flows which are not in the original code</td>
<td>34</td>
</tr>
<tr>
<td>2.8</td>
<td>Read access to SQL tables in analyzed applications.</td>
<td>40</td>
</tr>
<tr>
<td>2.9</td>
<td>Write access to SQL tables in analyzed applications.</td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>Queries from example forum application.</td>
<td>48</td>
</tr>
<tr>
<td>2.11</td>
<td>Passe’s impact on request latencies.</td>
<td>50</td>
</tr>
<tr>
<td>2.12</td>
<td>Passe’s impact on request throughput.</td>
<td>51</td>
</tr>
<tr>
<td>2.13</td>
<td>Memory usage of Passe in comparison to normal Django.</td>
<td>52</td>
</tr>
<tr>
<td>2.14</td>
<td>Passe memory consumption as the number of views varies.</td>
<td>52</td>
</tr>
<tr>
<td>2.15</td>
<td>Incorporating caches in Passe.</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>Pseudocode for hyperbolic caching in our framework.</td>
<td>64</td>
</tr>
<tr>
<td>3.2</td>
<td>Caption</td>
<td>66</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.3 Poor performance of LFU on dynamic workload.</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>3.4 Expiration-aware caching performance on synthetic workload.</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>3.5 Caching performance impact of perfect windowing.</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>3.6 Performance when modifying initial priorities.</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>3.7 Miss rate improvements on Memcached traces.</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>3.8 Caching performance of ARC and SPC traces.</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>3.9 Performance comparisons on synthetic workloads.</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>3.10 HC compared to LRU on Zipfian workloads of varying skew.</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>3.11 Miss rate’s impact on NodeJS throughput.</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>3.12 Per-class cost caching in the presence of transient loads.</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>3.13 Performance of HC as sampling size varies.</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>3.14 Performance of HC using retention-based sampling technique.</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modern web applications are both increasingly ubiquitous and increasingly important to their users. These web applications are varied, too—applications as different as photo sharing, social networks, e-commerce, and financial services are all implemented as web applications. And as the numbers of users of these applications grow and the applications become responsible for more and more sensitive data, the designers and developers of these applications must confront some of the classic problems of computer systems. Increasing traffic to these applications requires high performance from application servers and demands an increased cost for more server resources. On the other hand, the importance of the data stored in these applications demands that the confidentiality and integrity of that data remain uncompromised, even as the applications face a wide range of threats in the web environment. Overcoming these challenges requires increasing sophistication in the design, implementation, and deployment of web applications.

Unfortunately, despite the importance of securing web applications, applications today are susceptible to frequent, sometimes high-profile [80, 83], break-ins that ultimately compromise user data. Many times, a security failure in a single faulty part of the application exposes data from other parts. In other cases, aberrant applica-
tion behavior can be exploited to release otherwise protected data [73]. Solving these problems as an application developer can be complex and burdensome.

Secondly, to improve the performance of web applications, these services aggressively cache data originating from a backing store. The wide adoption of Memcached [42] and Redis [76] (key-value caching), Guava [46] (local object caching), and Varnish [90] (front-end HTTP caching) speak to this demand, as does their point-and-click availability on cloud platforms like Heroku via MemCachier [61], EC2 via ElastiCache [5], and Azure via Azure Redis Cache [10]. However, today’s caching algorithms are not tailored to web applications, and therefore significant performance gains may still be achieved through a more tailored approach that can incorporate properties such as item request frequencies, the costs of fetching items, or item expiration times.

In this thesis, I will demonstrate how the typical structure, implementation, and usage of web applications can be exploited to provide better application security on the one hand, and better application caching performance on the other. In the first case, I present the design and implementation of a web framework which automatically applies the principles of isolation and least privilege to server-side application code [15]. In the second case, I show how cache eviction strategies can be tailored to web application workloads, increasing application throughput without requiring modification of application code [16].

In Chapter 2, I discuss in more detail the security challenges web applications face, and present the Passe system which helps mitigate those problems. In Chapter 3, I discuss why current cache eviction strategies are not sufficient, and present the Hyperbolic Caching system. In the remainder of this chapter, I will explore why the web applications of today are designed as they are, and how that design enables us to tackle the old problems of security and caching performance with new techniques.
I will then detail the assumptions we make about how modern web applications are structured and how those applications behave under typical usage.

1.1 Structure of Modern Web Applications

Naively, a developer could construct a web application as a monolithic application, much like the classical design of a networked service. This single application would then be responsible for the whole process of responding to each HTTP request—receiving user web requests from a socket, parsing the HTTP headers, calling a function to construct a response, and writing the response back to the socket. If data needs to be maintained between requests, the application would store it in global variables. However, modern web applications are not structured this way.

At a high level, modern web applications employ the multi-tiered architecture shown in Figure 1.1. Application clients communicate with a load balancer or reverse proxy which routes the request to a server in the application or web tier. Servers in this tier execute application code without persistent state and operate independently of other application servers. All state persistence and server coordination is done through interaction with the storage tier and cache tier. Why are web applications structured this way, when a monolithic design is so much simpler? Because while
the monolithic design will work perfectly well for small workloads, it is difficult or impossible to scale such a system up to support even thousands, let alone millions, of users. To scale to such a degree, web applications need an architecture with more separation of components, and the modern multi-tiered architecture fulfills that need. In the rest of this section, I will describe how this architecture enables application scaling first and foremost by partitioning components into stateless and stateful. I will then describe how web frameworks ease the burden of developing multi-tier applications.

1.1.1 Separating State from Application Logic

In a multi-tiered web deployment, components are be divided into stateful and stateless components where state is defined as any information that persists between requests. The application tier is stateless, while the storage tier and cache tier are stateful. This separation is an almost fundamental requirement of scaling.

In order to scale, an application deployment must be capable of servicing multiple client requests in parallel. However, if state persists between requests, in order to ensure that the state is being correctly read from and written to, the application must perform some kind of concurrency control. Embedding this concurrency control into application logic would be cumbersome to developers. Instead, databases in the storage tier can provide this functionality. Now, because the application tier is a stateless component, it can readily scale to handle more requests. Calls to the storage tier can be made asynchronously, so that the CPU-intensive tasks of the application tier (e.g., parsing JSON inputs or rendering HTML templates) can continue while waiting for IO operations. Furthermore, the application server can spawn multiple instances of the application code, all interacting with the same storage tier, allowing the application to take advantage of multi-core processors. And finally, the servers themselves can be replicated.
In addition to allowing an application to serve multiple requests simultaneously, separating an application into stateless and stateful tiers segregates CPU-intensive application logic from IO-intensive storage. This allows the application developer to respond to application specific bottlenecks by scaling the two tiers independently. For example, if an application was not very computationally expensive, but served requests from a very large database, the storage tier could expand to hold more data by sharding the database across multiple servers. Alternatively, if the application was bottlenecked because each request required rendering and compiling an expensive templating system, more application servers could be installed without requiring any modification to the storage tier. In this way, resources can more easily be allocated to specific bottlenecks.

1.1.2 Easing Data Separation with Web Frameworks

Web frameworks help developers separate application and data-handling logic by providing libraries to interact with data storage and user sessions. Many popular web frameworks such as Ruby on Rails, Django, and CakePHP, employ a particular design pattern known as Model-View-Controller as their approach to separating application logic from data and formatting code. In this design pattern, model objects provide an abstraction for interactions with data, view objects define how those models are displayed to end users, and controllers provide application logic. The specific interpretation of this design pattern varies with different web frameworks. For example, in Django [32], views and controllers are combined into view objects and templates. View objects interact with models to implement any application logic and use templates to render HTML outputs (Figure 1.2). A request handler determines which view object should handle particular HTTP requests.

Importantly, this separation of data and application logic allows an application’s deployment to change with relative ease. Switching between a testing environment
with a SQLite backing store and a production environment with sharded databases requires simply reconfiguring the libraries supporting the application’s models. Similarly, introducing caching between the request handler and the view object or between the view object and the database can be achieved without modifying the code of the view object. We will exploit this separation to achieve our goals of improved security and caching performance.

1.2 Usage and Behavior of Modern Web Applications

In this thesis, we exploit both the regularity in the structure of web applications and the behavior of those applications in practice. To this end, we make assumptions about how applications interact with their data and how user interactions with those applications manifest in regular data access patterns.

Components Have Regular Data Flows While modern web applications are implemented with some separation between application logic and data-handling code, the applications still interact with that data in regular patterns. In particular, while a backing database may support and allow an expansive range of data queries and
modifications, a particular application component will only ever use some subset of those queries. Furthermore, because developers are specifying how particular requests interact with data, the flow of data and the ordering of data operations should also follow regular patterns. For example, a page which displays the logged-in user’s messages should always issue a request to get the logged-in user and then subsequently make a request for messages belonging only to that user. While data and application code does not explicitly declare these allowable sets of flows, modern web applications generate these kinds of regular data-flows implicitly.

Of course, applications may be written which do not have regular data flows. For example, an application that directly executes a given SQL query from an end user is clearly capable of generating arbitrary data flows. However, we believe that these applications are anomalous and choose to focus our attention on cases where the developer attempts to allow only specific data-flows.

We exploit this regularity in our secure web-framework by whitelisting expected data-flows. For caching, we will use the regularity of data access to detect any shifts in the costs of cache-misses. For example, if an item becomes more expensive to fetch, it may indicate that its backing store is under an unusual load and other items sharing that backing store are now more expensive as well.

Users Generate Regular Access Patterns. Modern web applications service requests from millions of users. Because the users act mostly independent of one another, application data access patterns exhibit a kind of independence. In particular, data accesses appear memoryless, in that the approximate probability of a particular item being requested is roughly equal to its relative popularity. Furthermore, these relative popularities can be described with Zipf-like distributions [18]. This is in contrast to the data access patterns of traditional applications, where a single client’s data accesses exhibit properties like access locality or repetitive uses.
Chapter 2

Protecting Web Applications with Passe

2.1 Passe Overview

Modern web applications play a central role in users’ online experiences. In doing so, these services often gather significant amounts of valuable, user-specific, and sometimes privacy-sensitive data. Unfortunately, despite the importance of this data, client-facing applications are susceptible to frequent break-ins that ultimately compromise user data [80, 83]. Many times, a security failure in a single faulty part of the application exposes data from other parts. In other cases, aberrant application behavior can be exploited to release otherwise protected data [73].

Mitigating these threats is typically not easy. Conventionally, an entire web application runs with one privilege, and often in one shared-memory address space. This has problematic security implications, because attacks that manage to overcome one portion of the application may affect its entirety. Even if application components can be better isolated by running with limited privileges and in separate processes, as discussed in Chapter 1 applications typically rely on shared persistent storage.
An attacker could exploit this storage as a communication channel. In this manner, attackers can target the “weakest link” of an application (which may undergo less scrutiny by developers) and then escalate their control. For example, breaking into a website’s public forums can lead to access to sensitive user data and passwords, either through explicit database queries or by directly accessing values in program memory. In some cases, attackers need not even compromise application code; unexpected application behavior can lead to execution paths which ultimately leak or compromise user data [73].

To deal with these threats, we exploit the typical design and behavior of web applications and apply three design principles. First, we split portions of application code into isolated components along natural isolation boundaries, taking advantage of the typical “switched” design of web applications. Second, in applying the principle of least privilege [78], we minimize the amount of privilege given to each component to only that privilege which the component needs to execute at that specific time. Finally, we use dynamic analysis to infer each component’s required privilege, such that the principle of least privilege can be largely automated.

While the principle of least privilege and the goal of maximizing isolation between components are old concepts, modern web applications provide a unique opportunity to apply these concepts. Automatically partitioning traditional, single-process applications is notoriously difficult [19,22]: they are typically designed with large amounts of shared memory and application traces can be long, with many user interactions intertwined in the execution trace. However, today’s scale-out architectures and their client/server division-of-labor offer new possibilities. They encourage developers to write server-side applications with components that offer narrowly defined interfaces and handle short requests. While this often leads developers to design their applications to support isolation, these applications usually all run in a single privilege
domain and address space. We, however, leverage these properties to automatically decompose applications into isolatable components.

This chapter presents the Passe project, a system which realizes these design principles in a typical client-facing network application and allows for the enforcement of learned security policies in a large-scale datacenter architecture. Passe runs developer supplied applications as a set of strongly isolated OS processes, though the design also supports running each component on a separate machine. The isolated components, or what we call views, are restricted to what data they can access or modify. Passe protects data by limiting views to particular data queries. For example, a view which only handles displaying user messages will be restricted to only making queries which fetch user messages. These queries are restricted further by applying integrity constraints to capture and enforce the data and control-flow relationships between queries and other data sources in the system. If the argument to the above query is always derived from the “current user”, then a data-flow constraint would assert that “current user” is the only valid argument to the fetch message query. If a permissions check is always executed (and returns True) before the fetch, then a control-flow constraint would assert that a permission check must be executed and must return True prior to the fetch query.

To discover the constraints for these queries, Passe monitors normal application behavior during a trusted learning phase. During this phase, our analysis engine not only learns which views make which database queries, but it also infers data- and control-flow relationships between database query results and later database queries. Data-flow relationships are defined as the relationship between a data source that supplies data to ultimately be used in a data sink—in Passe, a data source would be a database query result or a user-supplied input, and an associated data sink would be a query argument which uses that data. A control-flow relationship appears when a data source affects whether or not a particular query is executed at all. Passe captures
these relationships when the dependency is one of equality or set-membership, i.e., the data source supplies an unmodified input to the data sink. While more limited than general control-flow or data-flow dependency tracking, this approach captures relationships based on object identifiers, which are how applications typically express security policies (e.g., a particular message or post is associated with a particular set of allowable user IDs). Further, by restricting the set of relationships we enforce, Passe avoids a problem where most objects in the system are control-flow or data-flow dependent, even though they may only be “weakly” dependent. (This may occur due to over-tainting, a common problem in taint-tracking systems where cautious taint propagation rules generate excessive or spurious data-flow relationships.) Ultimately, Passe’s learning phase outputs an inferred policy. These policies are capable of capturing direct data-flow and control-flow dependencies between query results and subsequent queries. For example, an application may use two queries to implement an access control: the first query checks whether the current user is in the set of authorized users, and the second query only executes if the first query returns true. Passe would enforce this by requiring that the first query always return true before the second query could ever be issued.

The Passe analysis phase is related to work in Intrusion Detection Systems (IDS) [12, 56], which similarly analyze the behavior of applications to infer the “normal” behavior of the application. Unlike prior work in IDS, however, Passe translates these inferred relationships into integrity constraints which the runtime will later enforce. This translation from dependency relationships to integrity constraints is exactly what enables Passe to support rich data policies in a large-scale application architecture. Our analyzer may in some cases make inferences which are too strong, leading to some normal application functionality being denied. In this sense, Passe is a default-deny system: if particular queries have not been witnessed by the analyzer, then those queries will not be allowed. Developers can fix overly-strict constraints
by either adding test cases to correct Passe’s inferences or by modifying the policy constraints manually.

While it may be a source of developer frustration, we believe such behavior has additional security benefits. The history of web application break-ins shows that applications are too often written such that, even without a remote code execution exploit, attackers can make database reads or writes that are inappropriate given what the developer actually intended [73]. Because the application’s testing phase forms the basis for policy generation under Passe, it can serve as a check for such intent and helps prevent aberrant program behavior leading to data policy violations. Interestingly, code analysis techniques like symbolic execution, in finding the program’s exact constraints, would not provide such a feature.

We built a prototype of Passe on Django, a framework for building database-backed web applications in Python. Our prototype’s analysis engine runs unmodified Django applications and infers a policy configuration for the applications. This configuration specifies (i) how to split Django application into separate views, each running in their own sandboxed OS process, (ii) how to limit access for each view’s database queries, according to the principle of least privilege, and (iii) what integrity constraints to place on these queries. Our design was not specific to Django, and we expect the same mechanisms could be built into other popular frameworks.

We evaluated Passe’s effectiveness and ease-of-use by analyzing and running 11 off-the-shelf Django applications. We found that Passe can both restrict the set of queries each component can make, and infer richer application security policies through data-flow dependency relationships. We also evaluate the performance of our prototype on a series of application tests, measuring an increase in median request latency of 5-15 ms over normal Django, mostly due to the cost of data serialization between Passe’s isolated processes. While workloads served entirely from memory suffer a
37% drop in throughput, workloads requiring database interactions, as is typical for web applications, experienced a throughput reduction of about 25%.

2.2 Security Goals and Assumptions

The goal of Passe’s analysis is to infer the strongest possible query constraints which may be applied to isolated views. There are several potential problems with this. First, if an application is not easily decomposed, then Passe will fail to separate it into views, or if single views are responsible for large portions of the application’s behavior, the provided isolation will not be particularly useful. Second, if database queries are highly dynamic, Passe may not allow the queries at all. If queries are not protectable through simple data-flows (as one might expect in very complex applications), then Passe will not provide protections. We developed our prototype of Passe to explore how well these goals can be achieved with common web applications.

2.2.1 Threat Model

Passe assumes that application developers supply non-malicious, although possibly exploitable, application code to our framework, which runs on one or more application servers. This possibly faulty code may also include many third-party libraries. Thus, we do not trust that applications or their included libraries are secure. An attacker can exploit bugs in application views with the goal of compromising other views or shared data. Additionally, in the web setting, an attacker can return scripts to a client’s browser which attempt to access or extract information from other views (this includes traditional cross-site scripting attacks).

We do, however, assume that attackers are unable to compromise the trusted components of Passe. Further, we trust the underlying data store, the OS running our components, and, for web browser sandboxing, we trust that browsers correctly
enforce the new sandboxing features in HTML5. While these components may and do have bugs, they can be patched and updated. As a common platform shared by many websites, we believe there are greater incentives and opportunities to secure these components, as opposed to securing each application. Similar to the operating system, securing the framework only has to be done “once” to benefit all its users. Further, Passe’s trusted components provide functionality which is much simpler than the actual application code.

2.2.2 Motivating Classes of Vulnerabilities

There are three classes of vulnerabilities common to networked applications—and web applications in particular—that Passe is able to mitigate. We further discuss how Passe mitigates these vulnerabilities in Section 2.7.

1. **Poorly understood application behavior.** Even while using frameworks which prevent vulnerabilities in data operations (such as SQL queries), application developers may use library calls which have surprising behavior in unexpected settings. For example, the 2012 Github / Ruby-on-Rails vulnerability was caused by the default behavior of the Rails mass assignment operation. This operation allows web requests to set arbitrary attributes of an UPDATE query.

2. **Cross-Site Scripting (XSS).** A client’s web browser presents a possible channel to attack Passe’s isolation of views. Traditional XSS attacks may allow a vulnerability in one view to make AJAX requests to other views. For example, user input on a forum page is not properly sanitized, allowing one user to upload JavaScript which, when executed by another user, has malicious effects such as changing the second user’s password or accessing their sensitive data. Additionally, a compromised server-side view could use XSS as a side-channel
to defeat the server-side isolations of Passe. While numerous approaches exist to filter user inputs, discover vulnerabilities, and help translate applications to use proposed W3C Content Security Policies features, these techniques either cannot find all XSS vulnerabilities, or they require programmer effort to modify JavaScript code. Passe is able to mitigate many of the effects of XSS attacks using the same isolation model that it applies to application views.

3. **Arbitrary Code Execution.** Even when applications are programmed in high-level languages such as Python, there are occasional vulnerabilities allowing attackers to execute arbitrary code. While these problems may be infrequent, they are particularly damaging. For example, a vulnerability in the Python YAML library enabled attackers to gain complete control of an attacked Django runtime [71].

### 2.2.3 Security Properties of Passe

In the event that an attack against a particular view succeeds, Passe continues to provide the following security properties:

**P1: Isolation of Execution.** An attacker is unable to inspect or alter the memory of other application views. This provides security for the execution of other views. In the context of web applications, this applies to cross-site AJAX requests: only application views which normally communicate using AJAX are allowed to communicate during the secure execution mode.

**P2: Isolation of Data.** An attacker is unable to read or modify portions of the durable data store that are not accessed by the compromised view during its normal (i.e., correct) execution. For example, if the application is an online store with an attached discussion forum, and an attacker compromises only the forum view, he would still be unable to read or modify data associated only with the store’s functionality.
P3: Enforcement of Data Policy. An attacker is unable to violate high-level application data policies, even when the data concerned is normally accessible by the compromised view. For example, a correctly-behaving view may only fetch messages for end users that are “logged in” to the service, but because different users are logged in at different times, the view has high-level access to the entire set of messages. Even so, Passe ensures the finer-grain application security policy: even once comprising the view, an attacker cannot read messages of users that are not logged in. More specifically, Passe preserves data-flow dependencies on database query arguments and control-flow dependencies between queries. If, during a normal execution, a particular database query argument is always the result of a prior database query or the user’s authenticated credentials, then even in the case of a compromised view, that argument must still come from that source.

2.3 Passe Design

The design of Passe’s runtime accommodates the typical tiered, scale-out architecture of most client-facing datacenter services, illustrated in Figure 2.1. In this architecture, a request is forwarded to an appropriate machine in the service tier. The service tier (also called the application tier) is comprised of multiple machines, possibly running the same application code, which access shared storage through the storage tier. The storage tier is comprised of possibly many machines, handling separate, potentially replicated partitions (or “shards”) of the shared storage.

In Passe’s runtime (Figure 2.2), applications are decomposed into isolated views, running in separate sandboxed environments. This can be achieved through OS-level mechanisms, such as AppArmor [6], or by running views on entirely separate machines. Each of these views is responsible for handling specific requests which a dispatcher will forward. Passe introduces a stateless proxy between the service
and storage tiers which interposes on data queries. This trusted proxy approves or
denies data queries based on a set of constraints. These constraints are applied to the
supplied queries and a supplied token, ensuring that application data policies remain in effect even during a compromise of the application.

Passe provides an analysis system which monitors the “normal” execution of applications, and during this, learns the data-flow and control-flow dependencies of the application. This learning process occurs during an explicit testing or closed deployment phase, during which we assume components are not compromised.

2.3.1 Interacting with a Shared Data Store

Passe provides data isolation between application views. If two views never share data through the shared data store, then the compromise of one view should not affect the data of the other. While secure, enforcing strict data isolation greatly limits the type of applications one can build—applications frequently need to share data to provide basic functionality. For example, a user’s profile data may need to be shared between nearly all of an application’s components. However, all of the application components should not have unfettered access to this table.

In Passe, we only allow application views to interact with a shared data store through a query interface. Conceptually, an unbound query (as we will see later, in SQL these are query strings) has a set of arguments. Normally, when an application issues a query, it supplies an unbound query and a tuple of argument values. For example:

\[
\text{result} = \text{fetchUserMessage}(\text{uname} = "Bob")
\]

In order to enforce data policy, Passe must constrain the arguments to queries. However, these arguments are not necessarily hard-coded constants and may instead derive from prior database results. For example, a view displaying all of a user’s friends’ updates might issue two queries:
Figure 2.3: Example queries demonstrating the types of data-flow and control-flow dependencies. Queries take an argument set and a token as input.

```python
friends = fetchFriendsOf(uname = "Bob")
updates = fetchUpdates(author in friends)
```

Here, data from the first query is used as an argument value in the second query. Passe will attempt to enforce this relationship: that the second query should only contain arguments from the first query. In fact, this example demonstrates a data-flow dependency. In a data-flow dependency, the argument value of one query is equal to the result of a previous query. Another type of dependency is the control-flow dependency. In this case, the result of one query affects whether or not a second query would even be issued. Passe captures data-flow and control-flow dependencies which can be expressed with equality relationships. Figure 2.3 shows example application code demonstrating dependencies.

### 2.3.2 Protecting the Shared Data Store

Passe employs two mechanisms to enforce dependencies: a **database proxy** and **cryptographic tokens**. Every response from the proxy includes a token. This token is a set of key-value pairs which encode results from previous queries. Every request to the proxy must include a token, which the proxy will use to check that particular data dependency relationships are met (and that application code is not
trying to issue unauthorized queries). This token allows the database proxy to track what information has been returned to the view while remaining stateless itself (particularly important if the system employs multiple such proxies). In order to prevent compromised code from altering this token, it is cryptographically MAC’ed by the proxy. The key used for this operation is shared by Passe’s dispatcher and proxy. To prevent replay attacks with this token, the dispatcher and proxy include nonces in each token and track the nonces which have already been used.

In order to approve a query, the database proxy consults an access table and applies a two-stage verification process. In the first stage, the proxy checks whether the requested unbound query is whitelisted for that particular view. If not, then the request is denied (in this sense, the proxy is fail-safe). In the second stage, the proxy checks if the set of constraints associated with the unbound query is satisfied by the supplied argument values and token. Figure 2.3 displays a set of constraints for the associated unbound query.

2.3.3 Learning Constraints

Passe infers access-table entries during a learning phase. In this phase, Passe uses dynamic taint tracking to learn the data and control-flow dependencies experienced during the normal execution of application code. The developer can either supply test cases or run an “internal beta”. Once this phase has completed, Passe translates dependencies into the token-based constraints which will form the access table entries. These inferences would allow any of the witnessed traces to run, but then errs on the side of strictness. If the analyzer provides too strict of a configuration, the developer can either increase the number of test cases or alter the configuration manually.
Figure 2.4: Each arrow represents a communication channel in Passe’s runtime. Solid lines are trusted; dashed are untrusted. Blue lines correspond to developer-supplied code, which is embedded into Passe’s runtime components. When a request arrives at the dispatcher, it first gets processed by any number of middleware modules (which includes the session manager), before getting matched against the URL Map for dispatching to the appropriate view.

2.4 Passe Runtime in Web Setting

We implemented Passe as a drop-in replacement for Django, a popular Python-based web framework which relies on the “model-view-controller” design pattern discussed in Chapter 1. In this setting, the Passe runtime (shown in Figure 2.4) involves the following components:

- The **Dispatcher** uses the URL of a request to decide which view will handle a particular request.
- The **Session Manager** handles mapping user cookies to stored sessions.
- The **Authentication Manager** checks users credentials (username and password) and associates the current session with that user.
• The **Database Proxy** mediates access to a SQL database.

• The **View Server** provides a wrapper around the view function for handling inter-process communication and automatically binding tokens to requests in and out of the view. The view function itself is the unmodified developer code.

### 2.4.1 Isolating Views

In automatically decomposing applications into isolated views, we must solve four problems, related to (i) determining view boundaries, (ii) translating function calls into inter-process communication, (iii) dealing with global variables, and (iv) sandboxing these processes.

Passe determines application boundaries by leveraging the design of Django. In Django, application developers specify a mapping of requests to functions which handle the logic and HTML rendering of those particular requests. In Passe, we treat each of these functions as a separate view, such that each view is responsible for handling a complete request.

Passe must translate what were previously simple function calls into inter-process communication. Passe wraps application code with a view server, which handles marshalling function calls into this inter-process communication. This uses the Pyro library for Python remote objects, which automatically serializes the arguments of remote procedure calls using Python’s `pickle` module. The deserialization process is unsafe: if any object can be deserialized, then arbitrary code may be executed. This is dealt with by modifying the deserialization code to only instantiate objects of a white-listed set of types.

Because application code now runs in separate processes, previously shared global variables are no longer shared. However, in order to support a scalable application tier, developers are encouraged to share global variables through a narrow interface by modifying values in a single request state object. In Passe, changes to this object
are propagated back to the dispatcher by our view server. In order to minimize this
overhead, Passe computes and sends a change-set for this object. The dispatcher
checks that the change-set is valid (e.g., a view is not attempting to change the
current user) and applies it to a global request object.

Passe sandboxes these views by engaging Linux’s AppArmor and creating specific
communication channels for the processes. Each of the views communicates over spe-
cific Unix domain sockets with the dispatcher and the database proxy. As each view
server starts, an AppArmor policy (which Passe defines) is engaged, and the view
server becomes sandboxed. This prevents views from making system calls, communi-
cating over the network, or reading from the filesystem. Views may only read from a
limited set of files required for their execution. This set of files includes the Python
libraries and the application source code, allowing the view to read and execute those
files. When executed, these files run within the view’s sandbox. Network access is
limited to the Unix sockets used to communicate with Passe’s components.

2.4.2 Constraining SQL Queries

Applying Passe constraints to SQL queries requires two mechanisms. First, we need
to specify how a SQL query maps to our notion of an unbound query. Second, we
need to specify how SQL query results are stored and referred to in the token.

In Django, applications issue queries as a query string and an ordered list of
arguments. For example, a view might supply a query string

\[
\text{SELECT (text,fromUser) FROM msgs WHERE toUser = ?}
\]

and an argument “Bob”. For Passe, we treat the query string itself as the unbound
query. In the access table, we store the strings with a cryptographic hash of the query
string (to reduce the required storage).
In order to store query results in the token, we again use a hash of the query string to refer to the “source” of the data. In addition, we separate the results by column so that Passe can place constraints at the column granularity, rather than the entire result set.

2.4.3 Handling a Web Request

When a view receives a request, the dispatcher gives it an initial token containing the current user (or “anonymous” if appropriate) and any HTTP request variables (query strings and header fields).

The contents of Passe’s tokens during the execution of a view are shown in Figure 2.5. As the view makes requests into trusted Passe components, those components respond with updated token values, which may in turn be used in future requests. Whenever updating a token, the trusted component generates a new message authentication code (MAC) covering the token’s latest contents.

This example demonstrates how Passe’s constraints operate in practice. When operating correctly, this view displays all posts from the current user’s friends, in this case Alice’s friends. From a security perspective, the view should only be able to enumerate the user’s friends and only read those updates from her friends (confidentiality properties). In more detail, the view initially receives the HTTP request object and a token containing the current user. The view makes two database queries, each with an argument protected by a data-flow constraint. The first query derives its argument from the “User” key. The second query’s argument, however, is derived from the results of the first (and matches a key that names the first query and the “uid” column). These constraints ultimately enforce the application’s desired policies, even if the view were compromised: the view can only see updates for users contained in the result set of the first query and that query can only return the current user’s friends.
HTTP Request, T1 = Token{{User, “Alice”}}

Query = “SELECT * FROM friends
WHERE friendsWith = ?”
Args = [“Alice”], Token = T1

Results = [“Bob”, “Charley”]
T2 = Token{{User, “Alice”},
(q1hash_uid, [“Bob”,...]) }

Query = “SELECT * FROM updates
WHERE author in ?”
Args = [set(“Bob”, “Charley”) ], Token = T2

Results = [“foo”, “bar”]
T3 = Token{{User, “Alice”},
(q1hash_uid, [“Bob”,...]),
(q2hash_post, [“foo”,...]) }

HTTP Response, Token = T3

Figure 2.5: At the start of a request, Passe provides a view with an initial token containing the current user. As the view makes database queries, it supplies the current token which Passe’s proxy uses to check the integrity constraints on the arguments supplied to the query. The database proxy replies with updated versions of the token based on the query results.
2.4.4 User Authentication and Session Management

Passe’s constraints specify a relationship between a query and the trusted sources that supply the values to that query (or affect the control-flow leading to that query). Certain token keys—such as the current user and HTTP request parameters—do not originate from prior queries, however, but rather serve as a foundation for the constraints of a view’s queries. It is vital that the mechanisms used to generate these tokens are sound: If an adversary can forge the current user, confidentiality is largely lost.

Traditionally, Django has two mechanisms for associating a request with a user. Either a view can explicitly call into the authentication library which returns the associated user, or the request is part of a session already associated with a logged-in user. In the latter case, before a view handles a request, the dispatcher calls into the session manager, which reads the request’s session cookie and checks whether it is already associated with a user.

In Passe, we modified these two mechanisms so that both session and authentication manager will securely embed the current user’s ID in a token, rather than simply returning the user to the dispatcher or login view, respectively. This change also entails that these managers know the shared symmetric key used to MAC tokens. To prevent a compromised view from stealing session cookies, the Passe dispatcher elides session information from the token before forwarding the request to the view.

2.4.5 Isolating Views at a Client’s Browser

An end user’s browser presents a significant attack channel for an attacker with control of a view. The attacker can return an HTML page with malicious code used

\footnote{In fact, our implementation could have left the session manager unmodified, as it only communicates with the dispatcher, which could have embedded the user ID on its behalf. Because the authentication manager is accessed by an untrusted login view, however, it must implement this secure token embedding itself.}
Figure 2.6: An attacker who has compromised view A is unable to directly query the database for view B’s data. However, by returning a script to the user’s browser, the attacker can exfiltrate B’s data by having the browser make seemingly normal requests to view B.

to circumvent least-privilege restrictions and thus access other portions of the web application. For example, if an attacker compromises a view A which cannot access restricted portions of the database, the attacker can return JavaScript which loads and scripts control over another view B. View A can then use the results of view B to gain access to otherwise inaccessible portions of the database, as shown in Figure 2.6. This attack is similar to XSS in that the same-origin policy fails to protect other portions of the web application from the malicious script. Typically, applications prevent XSS attacks by filtering user inputs. However, an attacker with control of a view can circumvent these protections by inserting JavaScript directly into a response. Even when an application uses a feature such as Content Security Policies (CSP), entire domains are typically trusted to supply scripts [26].

To mitigate this cross-view attack channel, Passe supports execution with isolation even at the client browser. In particular, to preserve isolation between views, Passe’s dispatcher interposes on AJAX requests between views. The dispatcher keeps a mapping, learned during the Passe training phase, of which views are allowed to
originate scripted requests to other views. Based on this mapping, the dispatcher approves or rejects requests. This requires the dispatcher to know which view originated a particular request and whether that request is an AJAX request. The dispatcher derives this information from two HTTP headers: \texttt{Referer} and \texttt{X-Requested-With}.

To prevent adversaries from circumventing these checks, Passe must ensure that an attacker cannot remove or modify these headers. Towards this end, Passe sandboxes a view’s HTTP responses using the new HTML5 \texttt{sandbox} attribute with iframes. Every view’s response is wrapped in this sandboxed environment by the dispatcher. We implemented a trusted shim layer which ensures that the headers are correctly added to each outgoing AJAX request. Our approach is similar to the shim layer used to enforce privilege separation in HTML5 applications as introduced by Akhawe et al. [3].

### 2.5 The Passe Analysis Phase

During the analysis phase, Passe monitors application execution with the following goals:

- **Enumerate Views.** Passe’s runtime only executes a fixed set of views. The analysis phase is responsible for enumerating these views and assigning each of them a unique identifier.

- **Enumerate Queries.** Passe associates each view with the SQL queries witnessed during analysis.

- **Infer Dependency Relationships between Queries.** The analysis phase is responsible for determining data and control-flow relationships between queries, prior query results, and other data sources.
• **Translate Dependencies into Enforceable Constraints.** Dependencies witnessed during the learning phase must be translated into constraints which the Passe runtime is capable of enforcing.

Passe’s analysis phase achieves these goals using a combination of taint tracking and tracing. As queries execute, Passe constructs an event log and adds taint values to the database results, and once execution has completed, Passe processes this log and outputs the allowed queries for each view, and the associated constraints on those queries. The analysis phase runs on a PyPy Python interpreter which we modified to support dynamic taint tracking.

### 2.5.1 Dynamic Taint Tracking in Python

In order to support dynamic taint tracking for Passe, we developed a modified version of the PyPy Python interpreter (our modifications are similar to approaches found in [24, 47, 79]). Our modifications allow for fine-grained taint tracking through data passing operations and some control-flow tracking. The interpreter exposes a library which application-level code can use to add taint to a particular object, check an object’s current taint, and check any tainting of the current control-flow.

Each interpreter-level object is extended with a set of integer taints. As the interpreter processes Python bytecode instructions, any instruction which returns an object propagates taint from the arguments to that object. Additionally, because many functions in PyPy are implemented at the interpreter level (and therefore are not evaluated by the bytecode interpreter), these function definitions also need to be modified to propagate taint. For our prototype implementation, only functions for strings, integers, booleans, unicode strings, lists, dictionaries, and tuple types were modified.

²Source code for Taint Tracking PyPy is available at https://github.com/kantai/passe-pypy-taint-tracking
In order to track control-flow tainting, the interpreter checks the taint of any boolean used to evaluate a conditional jump. If the boolean contains a taint, this taint is added to the current execution frame. Taints are removed when their originating function returns. In our prototype, the current control-flow taint does not propagate during data operations—if a control-flow taint is active while a data-flow operation occurs, the result is not tainted with the control-flow taint. While this causes the analysis to miss some control-flow dependencies, this is purely a limitation of the prototype, and the applications we tested were not affected. While including this feature will increase the possibility of over-tainting, Passe’s constraints only capture equality and set-membership relationships which mitigates many of the effects of over-tainting.

2.5.2 Tainting Data Objects and Logging Query Events

During the analysis phase, Passe creates a log of query events by tracing the normal Django execution of the application. In order to track the data-flow and control-flow dependencies in the application, these events contain taint tags for each logged data object.

In addition to capturing query calls, Passe must properly add taints to data objects as they flow through the application. As HTTP requests enter the application, Passe taints the initial data objects. Later, as each database query is made, Passe also adds taints to each result column.

When Passe captures a query call, it logs the event with the following information:

1. The view responsible for the query.

2. The query string.

3. An ordered list of the query’s argument values and the current set of taints for each of those values.
4. Any previous database results or initial data objects, and these objects’ associated taints.

5. The control-flow taint set for the current execution context. In addition to a set of taint tags, for each permissions library call which affects the control-flow, the name of the checked permission is included. The permissions library is special-cased because of the “root” permission set. (Existence of a particular permission may be checked, or if the user is root, then the check is always approved.)

This information will allow the analyzer to translate witnessed dependency relationships between queries and objects into the integrity constraints used by the Passe runtime. Dependency relationships here are captured by the included taint sets.

2.5.3 Inferring Constraints from Dependency Relationships

Knowing that specific database query calls depend on previous query results is not sufficient for the Passe runtime to enforce constraints. Rather, Passe collects the logged query events and uses these to infer enforceable constraints. To do this, Passe collects all of the events for a particular \((query \ string, \ view)\) pair and merges the dependency relationships.

The analyzer constructs constraints which are the union of all witnessed data-flow relationships: if, across all witnessed query events, multiple data-flow relationships exist for a particular query argument, then \(\text{any}\) of those sources will be allowed to satisfy that argument’s constraint. On the other hand, if in some query events, no data-flow relationships exist at all, then the argument will be left unconstrained. To capture data-flow relationships, the analyzer only checks for \(equality\) and \(set\ membershi p\) relationships. These two relationships capture relationships based on object identifiers, which are typically how applications express security policies. (For ex-
ample, a particular message or post is associated with a particular set of allowable user IDs.) As we will see in our example applications (Section 2.7), no policies were missed because of this limitation. By requiring equality, Passe mitigates many of the problems normally associated with over-tainting. Normally, if “too much” taint propagates in an application, constraints based solely on taints will be too strong. In Passe, however, both the taints and the values are required to be equal, which reduces the chance of over-constraining a particular query. In cases where this does happen, then the developer can include test cases where equality does not hold which will prevent that constraint from being inferred.

Passe captures control-flow relationships similarly. For each query event, Passe determines which control-flow relationships affected that event. Passe then creates a set of invariants for the query based on these relationships. Here, unlike in data-flow constraints, there is a higher chance of over-fitting the invariant. For example, the `getUID` query in Figure 2.3 (on page 19) affects the control-flow of `getData1`. Passe could infer an invariant containing an OR-ed set of user IDs. This invariant, however, is too strong in practice: it fits the invariants to precisely those witnessed during the testing phase. Thus, rather than unioning these sets of possible invariants, Passe takes the intersection of these sets to construct the final invariant. For this example, the invariant $R1 \in R4$ would be witnessed for all `getData1` events, while invariants such as $R1 = \text{Alice’s UID}$ would only be witnessed for a few events, and therefore would not contribute to the query’s constraints.

When this translation phase ends, each view will be associated with a set of allowable queries. Each of these queries will have an associated set of control-flow and data-flow constraints, which are exactly what the Passe runtime uses in its access table to enforce query integrity.
2.5.4 Monotonicity of Analysis

By design, the Passe analysis phase does not witness all possible code paths and, therefore, Passe’s inferences may prevent certain code paths from executing correctly. However, developers can increase the number of test cases witnessed by the Passe analyzer and increase the allowable code paths. In respect to this, Passe guarantees monotonicity: additional tests cannot *reduce* the set of allowable queries. To see why this is the case, imagine that Passe witnesses an additional query event for some \((\text{query}, \text{view})\) pair. If this event creates any new data-flow constraints, they only increase the allowable data-flows. If, previously, that particular query argument was unconstrained, then it will remain unconstrained (again, because data-flow constraints only add new paths). The same is true for any new control-flow constraints, because a control-flow constraint will only be added if it holds for all the witnessed events of a particular \((\text{query}, \text{view})\) pair.

2.5.5 Impacts of False Positives and Negatives

**False Positives.** Passe’s analysis phase is capable of both false positives and false negatives when detecting dependencies. False positives occur when the application developer wishes to allow data-flows which Passe’s analysis phase does not witness. This results in a policy which disallows those data-flows. The developer can resolve such false positives by including new test cases which cover those particular data-flows.

**False Negatives.** Passe is also capable of false negatives when detecting dependencies. In these scenarios, Passe will generate a policy which is too permissive, such that, in the event of an application compromise, an attacker would be able to execute data-flows which should be constrained. This can only occur if a witnessed dependency is not captured by our taint tracker. As discussed in Section 2.5.1, our
def view(select_group):
    group = get_group_A()
    if select_group == 'B':
        group = get_group_b()
    update_all(group)

def malicious_flow(select_group):
    group_a = get_group_a()
    # Ignore the value of select_group
    update_all(group_a)

(a) Original view code. (b) Allowable malicious flow.

Figure 2.7: Passe’s constraint system will allow some flows not in this example view’s code. For example, a malicious attacker who has taken control of the view could then execute the flow in subfigure (b).

prototype can fail to detect certain kinds of control-flow dependencies. A developer can remedy such missed dependencies by manually inserting those dependencies into the outputted policy. While this is an unsatisfying method, we did not encounter any such cases of false negatives in our tested applications. This is also a current limitation of our prototype, rather than Passe’s underlying method; a more complete implementation of taint-tracking in the Python interpreter would not encounter false negatives while detecting dependencies.

2.5.6 False Negatives from Intersecting Dependencies

As discussed in Subsection 2.5.3, Passe only infers constraints that are satisfied in all data-flows through a query (i.e., an intersection of satisfied dependencies, rather than an ORed set of ANDed dependencies). This means that Passe’s control flow constraints only check whether a given query is allowable given some control flow state. It does not check whether a particular argument should come from a given source in one control-flow or another source in a different control flow. This behavior makes Passe’s constraint system more permissive than a direct union of allowable flows—some non-witnessed flows would be allowable. In Figure 2.7, for example, two different flows pass through the `update_all` query. However, for this query, Passe only infers a data-flow constraint requiring that the argument must come from either `get_group_a`
or \texttt{get\_group\_b}. If an attacker compromises the view, she would be able to ignore the \texttt{select\_group} input.

This problem could be addressed in a number of different ways. For example, rather than constructing constraints by forming an intersection of satisfied dependencies, constraints could be constructed by unioning the satisfied dependencies for each flow. In this case, the \texttt{update\_all} query would now have two constraints, either of which must be satisfied:

\begin{verbatim}
arg in get_group_a OR (select_group == 'B' AND arg in get_group_b)
\end{verbatim}

Because this modification would make Passe’s constraint system more restrictive, developers may need to supply more test cases to ensure that they correctly captured all of the expected data-flows in their application. More restrictive system could attempt to explicitly encode the view’s data-flow in the Passe token which the DB proxy would then use to enforce strict adherence to witnessed data-flows. However, in the applications we studied, the developer’s intended data policies were correctly enforced with our more permissive scheme.

\section*{2.6 Implementation}

We implemented the Passe runtime and analysis engine as modified versions of Django 1.3\textsuperscript{3}. For the analysis engine, we modified certain Django libraries to make analysis easier—in particular, the authentication and database libraries—by adding annotating calls for the tracer. Further, we use our modified version of the PyPy 1.9 interpreter to perform our dynamic taint tracking.

For the runtime, we modified the Django dispatcher to support interprocess calls to views, and the database, authentication, and session libraries were modified to make proxied requests. A Gunicorn HTTP server just acts as the network front-end, 

\textsuperscript{3}Source code of the Passe framework is available at \url{https://github.com/kantai/passe-framework-prototype}
accepting HTTP requests before passing them to Django (and its dispatcher). Our database proxy provides support for two different database backends, PostgreSQL and SQLite.

In total, Passe required 2100 new lines of code (LOC) for Passe-specific functionality, as well as changing 2500 LOC in the Django library and 1000 LOC in PyPy. Our HTML5 sandbox requires 320 lines of JavaScript code, which are inserted into responses automatically by our dispatcher.

### 2.6.1 Decoupling Middleware from the Dispatcher

Django allows developers to use *middleware libraries* to add various functions to applications (e.g., page caching, user session management, or cross-site request forgery protection). These libraries register for callbacks before and after normal view calls.

In order to mitigate the damage of faulty middleware libraries, Passe runs middleware in the same manner as views. Namely, middleware libraries are wrapped in a server and, where in Django normal function calls are made, in Passe these are inter-process procedure calls.

### 2.6.2 Unsupported Django Behavior

While we attempt to provide an interface identical to Django, our modifications do require some changes to this interface: views are forced to authenticate users through the default authentication library, which we modified, applications cannot use arbitrary global variables and the URL Map may only contain views found during analysis.

**Custom Authentication.** Developers may attempt to authenticate users directly, circumventing the authentication library. In our system, this will fail, as only the authentication server is able to create a new token for the application. This is prob-
lematic for applications that use external sources for authentication (e.g., OAuth). Our prototype could be extended to support different authentication libraries, or to provide a generic API which would allow Passe to be integrated with custom authentication method. This, however, would still require modifying some applications to use our API.

**Global Variables.** Because views in Passe run in separate processes, global variables cannot be used to pass information between views. However, passing information through global variables is discouraged by the Django framework. Using global variables in this way can lead to problems even in normal Django deployments where multiple worker processes are responsible for processing concurrent requests in parallel. Because these workers do not share an address space, propagating information through global variables could lead to inconsistencies between requests. As such, none of the applications we tested used global variables in this way.

**Dynamic URL Map Modifications.** Django allows developers to modify the URL Map to add new views dynamically. While Passe could possibly be extended to support such behavior by giving the new view the same permissions as the parent view, this was not implemented. Instead, if a view attempts to modify the URL Map, it fails, as it has no access to the dispatcher’s URL Map object. We did not encounter this problem in any of the applications we tested.

**External Communication.** A normal Django application may use a host of features in Python to communicate externally through the filesystem and network sockets. Because of Passe’s AppArmor sandboxes, however, these features will not work as expected. However, none of the applications we evaluated attempted to use these features.
2.7 Protections for Real Applications

To understand how Passe executes real applications, we ran and tested eleven open-source Django applications, manually inspecting both the application source code and Passe’s inferred policies. We assessed the source code for instances of application security policies and, in particular, those impacting data integrity or confidentiality. We then checked whether Passe correctly enforces those policies. Across all applications, we found four instances of clearly intended policies missed by Passe, exposing the application to confidentiality violations if compromised by an attacker. However, each of those instances could be remedied through simple changes in application code.

In this section, we evaluate the following questions:

§2.7.1. How does Passe mitigate the three classes of vulnerabilities in our threat model?

§2.7.2. How difficult is it to construct end-to-end test cases for Passe to analyze applications?

§2.7.3. What coarse-grained isolation properties can Passe provide to applications?

§2.7.4. Are there security policies in these applications which Passe’s dependency constraints cannot enforce?

§2.7.5. Examining three case studies in more depth, how does Passe capture fine-grained security policies?

§2.7.6. How do common web application vulnerabilities apply to Passe?
2.7.1 Passe in the Presence of Vulnerabilities

Unexpected Behavior

When applications exhibit unexpected behavior, Passe is able to prevent the attacker from compromising the database in many cases. For example, in the 2012 Github / Ruby-on-Rails attack, a mass assignment vulnerability allowed users to set the UID for the public key being uploaded. This allows users to upload a key for any user. In Passe, the normal path of the code would create a constraint for the `UPDATE` statements asserting that the UID must be equal to the current account.

XSS Attacks

Passe can mitigate many of the effects of XSS vulnerabilities. Passe restricts the content returned by views to only making AJAX requests to a whitelisted set of other views. For example, if a view containing user-generated content (such as a forum page) does not normally call to other views, then no XSS vulnerabilities in the view will be able to call other views. While this does not prevent scripting attacks which do not make cross-site requests, it does prevent views from using scripts to open attack channels against unrelated views. This allows Passe to preserve isolation between views, even at the client’s browser.

Arbitrary Code Execution

Some web applications contain bugs which allow arbitrary code execution. For example, a commonly used YAML library in Django allowed arbitrary object deserialization, ultimately allowing remote code exploits [71]. YAML parsing libraries exposed node.js and Rails applications to a similar attack [70,72]. Passe mitigates the threat of this by restricting developer-supplied code to specific database actions.
Figure 2.8: For the 11 applications we analyzed, we measured the proportion of tables that can be read by at most $N$ views.

Unfortunately, an attacker who has complete control over a view can launch a phishing attack, displaying a convincing login screen to users. This is more damaging than normal phishing attacks as this page will be served by the correct web host. Therefore, it is still important to recover from an attack quickly, even though Passe continues to provide data security properties during an attack. Other similar systems are also susceptible to this attack, including those incorporating more formal mechanisms such as DIFC [44]. We expect Passe to be integrated into system monitoring and alerting, such that developers can quickly detect (blocked) attempts to circumvent our protections. These alerts are also useful when a policy is too restrictive.

### 2.7.2 Building End-to-End Tests

To perform end-to-end tests, we wrote a test suite for each application using Selenium, a library for automating web-browser actions. Our suites tested applications entirely through their web interfaces by performing normal application actions. After running an application through Passe’s testing phase, we ran the application in the secure runtime of Passe, and when the inferred constraints were too strong, we added more
Figure 2.9: For the 11 applications we analyzed, we measured the proportion of tables that can be read written by at most $N$ views.
Table 2.1: For each application, we measure the number of browser actions performed in our test suites for Passe’s analysis phase and the number of discovered views, queries, constraints, and higher-level policies. Code coverage numbers reflect what percentage of the application code was covered by our tests (both in runtime and analysis phases). The lines of code (loc) not covered were either completely unreachable or unreachable through web browsing alone.
tests to the test suite. Each of these test suites took less than an hour to construct and was comprised of about 200 lines of Python.

To understand how much testing was required for each application, we measured the number of browser actions in each of the test suites we developed. The table in Table 2.1 displays these measurements. An important note is that while each application required a large number of browser actions to test the application, many of these actions were performed on the \texttt{django-admin} interface. Because this is a standard interface, a more advanced testing framework could automatically generate tests for this interface, a possible direction for future work.

In order to run with JavaScript sandboxing on the browser, Passe requires that a mapping from AJAX requesting views to the responding views be constructed. To see how much additional burden was required to generate this mapping, we tested the most AJAX-intensive application in our test cases (\texttt{social-news}) in the JavaScript sandboxing mode. We modified our end-to-end test scripts so that elements would be selected in the sandboxed frame rather than in the parent window. Other than these changes, the original end-to-end tests we developed to capture query constraints were sufficient to capture all of the allowable AJAX requests as well.

### 2.7.3 Isolation Properties

To understand how much isolation Passe provides by restricting each view to the set of queries it needs, we measured the proportion of views that can access each table of an application (Figures 2.8 and 2.9). Half of our applications’ tables are readable by at most 7 views. Still, some tables can be accessed by nearly all of an application’s views. For example, in the blogging applications, the user table holds an author’s username. Because most of the views display authored posts, the user table can be read by most views. When we look at views with write access, however, the separation is much stronger. Fully half of the tables for all applications are writable by only one
or two views. Of course, these results do not speak to the guarantees provided by the
inferred constraints, which further strengthen the application’s security properties.

We measured the number of constraints Passe inferred for each application (Table
2.1). Additionally, we characterized higher-level policies by inspecting constraints by
hand, discovering that Passe successfully finds 96% of the 105 possible application
policies. Because we characterized these policies by hand, we cannot eliminate the
possibility that we incorrectly characterized or left out policies.

2.7.4 Fetch-before-Check Problems

It is important to understand the scenarios in which Passe can miss an application pol-
icy. In all of the applications tested, Passe missed four application policies (out of 105
total policies): one in each of the simplewiki, django-articles, django-profiles,
and the django-forum applications. In all four cases, Passe missed the policy for
the same reason: application code fetched a data object before evaluating a permi-
sion check. The simplewiki application, for example, performed read permission
checks only after fetching wiki pages from the database. While this behavior poses
no confidentiality problem if the application is never compromised, it is clearly not
enforceable by Passe. This breaks Passe’s assumption that the normal behavior of
the system does not leak information. (Still, such an implementation can be dan-
gerous from a security perspective: Even when not compromised, the application
fetches data even when permission is denied, which may expose the database to a
denial-of-service attack if the fetched content may be sizable.)

This behavior can be quickly remedied, however. We fixed Passe’s missed infer-
ences by moving the permission checks to precede the data queries. In each of these
applications, these changes required modifying fewer than 20 lines of code.
2.7.5 Case Studies

Social News Platform  

**social-news** is a Django application which provides the functionality of a social news service such as Hacker News or Reddit. Users can submit stories to one of many user-created topics and each of these stories supports a comment section. Users vote an item up or down, and the final count of votes determines an item’s score. A recommendation system then uses these votes to suggest other stories which a user might like.

The *social-news* application contains three queries which need to be modified for the application to run with Passe. The application constructs these queries using string replacement to insert arguments rather than using SQL argument passing. This is known to be a less safe programming practice, as it can expose applications to SQL injection attacks. However, in Passe, such bugs cause an application to fail *safely*, as Passe will simply deny the constructed queries which it does not recognize. In order for the application to run correctly, 5 lines of code were modified so that query arguments were passed correctly.

With these changes, Passe correctly finds and enforces 17 policies for the application. Most of these are data integrity policies. For example, only the author of a post is authorized to change the content of that post. However, the post’s score is an aggregation of other users’ votes and each user is allowed to contribute only one vote to this score. Passe captures this by constraining the queries which log the votes, and the queries which tally those votes. Constraints are applied on the up-vote and down-vote views. These views issue an insertion query which puts a new row into a link vote table, or an update query which changes the direction of the vote. Passe ensures that these queries are associated with the current user, and a database uniqueness constraint ensures that only one row exists per user. The application then updates the score field of the news item by issuing a **COUNT** query and a subsequent
UPDATE query. Passe ensures that the updated score value is exactly the result of the associated count query.

**CMS Platform**  *django-articles*, one of the applications we tested on Passe, provides a simple CMS platform that allows users to create and manage a blog. New articles can be added with a web interface, and it supports multiple authors, article groups, and static pages.

This CMS application, like many Django applications, includes and relies on the *django-admin* module. This module provides a full-featured interface for modifying, adding, and deleting models in any application. To support any significant set of Django applications, Passe must support the admin module, and it does. Passe is able to infer strong constraints for this module. The admin module makes calls to Django’s permissions library, and Passe infers control-flow constraints based on those calls. In the case of the CMS platform, the Passe proxy requires that a view present a token possessing the “Article.add” permission to add a new article to the database.

Passe additionally enforces integrity constraints on queries creating and modifying blog posts. In particular, Passe requires that a new post’s author matches the current user and that the content of that article matches the user’s request variables. Passe ensures that a user is allowed to post articles by checking that user’s coarse permission set.

**Web Forum**  We also tested a simple forum library *django-forum* under Passe. This application allows developers to run an online forum, which supports user accounts, user groups, and multiple forums with display permissions. To support creating new groups and forums, the application uses the default *django-admin* interface.

*django-forum* supports linking particular forums to user groups, such that a given forum is hidden to and cannot be modified by users outside of that group. In the application, this access control is implemented by (i) retrieving the set of groups for
the current user, (ii) issuing queries both to fetch those forums with a matching group
and those with a public (null) group, and (iii) fetching all threads to those forums.
Note that the application never explicitly declares the policy that users should only
be able to see threads in their groups; it is only implicit in the application’s execution.
Passe makes this policy explicit, and it is enforced by the database proxy as a data-
flow dependency (Figure 2.10).

The django-forum application also provides an example of a control-flow con-
straint. Before adding discussion threads to a forum, a view checks whether the
current user has access to the forum. This check involves a SQL query which fetches
the set of allowable users, and then, in Python, the view checks whether the current
user is in that set. If the user has access, the view fetches the appropriate forum’s
ID, and uses this ID as an argument for creating a new thread.

In this example, the first query is a control-flow dependency. Later queries do
not have arguments reflecting its return values, and thus these three queries do not
form a data-flow dependency. However, the Passe analyzer correctly determines that
the first query, along with the current user, has tainted the control-flow, and infers a
constraint that the current user must be contained within the set of allowable users
for the second and third queries to be executed.

2.7.6 Common Web Vulnerabilities and Their Effects on
Passe

Though Passe protects against more general exploits, it is important to understand
how various common web vulnerabilities are relevant to Passe and its protections.

**SQL Injection.** For the purposes of Passe, SQL Injection attacks are mitigated
by the use of Django, which strongly encourages developers to issue queries using
separate query strings and query arguments. For applications which do not use this
argument passing method, Passe will prevent these from causing SQL injection vul-
Figure 2.10: *django-forum* executes three SQL statements to retrieve the accessible threads for the current user. SQL2 is restricted to only return threads from forums discovered in SQL1 or SQL0. These assertions chain these queries together, enforcing their data-flow relationship.

nerabilities. This is because Passe’s database proxy expects that the only parts of a query allowed to change are the arguments to the SQL query. If the query changes from string manipulation rather than SQL argument passing, then the query will no longer be recognized as the “same” query witnessed during training, and Passe’s proxy will reject the query. This requires that the developer change their application and adopt the preferred approach.

**Cross-Site Request Forgery.** Django mitigates CSRF attacks by requiring forms to carry CSRF tokens, which are used to check that requests came from a legitimate web request. If an attacker compromises a view, they can always forgo this protection for that particular view. Worse, however, this attacker may be able to steal CSRF tokens and use them for other views. To mitigate this attack, we can associate CSRF tokens with particular views, and thus prevent a view compromise from affecting other views.

**Click Jacking.** An attacker may attempt to load web pages secured by Passe in a HTML frame and maliciously access the page using JavaScript. In order to prevent this attack both from external sites, and from an internal view which an attacker
has compromised, Passe adds the \texttt{X-Frame-Options} header to all outgoing responses. This prevents the web page from being displayed inside of a frame.

\section*{2.8 Performance Evaluations}

To evaluate the performance of Passe, we ran three applications in Django and Passe, measuring the time to fetch each application’s home page. Our testing system used Gunicorn 0.15 to dispatch web requests through WSGI, PostgreSQL 8.4 as the backing database, Python 2.7, and Ubuntu 11.04. Gunicorn is the default Python HTTP server on popular hosting platforms such as Heroku. Because Django and Passe do not require many of the features of Apache, lighter-weight servers such as Gunicorn may be used. Our test server had 2 Intel Xeon E5620 CPUs, with 4 cores each clocked at 2.40GHz. Because Passe’s database proxy pools database connections, in order to fairly compare the throughput with plain Django, the plain Django version used pgpool for connection pooling. (Without pgpool, Passe outperformed vanilla Django in throughput measurements.)

In order to understand Passe’s performance on real applications, we examine performance on the case study applications we detailed earlier. Further, to explore the base overhead Passe imposes on simple applications, we developed a benchmark application that immediately renders a response from memory without using the database.

\subsection*{2.8.1 Latency and Throughput}

We measured latency of requests by repeatedly fetching application pages with a single user on the JMeter benchmarking platform. Figure 2.11 plots the latencies of 1000 requests. While Passe’s latency overhead of 5-13 ms is not insignificant, applications and service providers often target much larger end-to-end latencies, e.g.,
Figure 2.11: Request latency for accessing specified applications’ pages. Error bars indicate the 90th and 10th percentiles.

<table>
<thead>
<tr>
<th>Application and Page</th>
<th>Number of Queries per Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>0</td>
</tr>
<tr>
<td>Forum</td>
<td>1</td>
</tr>
<tr>
<td>CMS</td>
<td>3</td>
</tr>
<tr>
<td>SocialNews-Home</td>
<td>4</td>
</tr>
<tr>
<td>SocialNews-New</td>
<td>4</td>
</tr>
<tr>
<td>SocialNews-Login</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Number of queries per response in the applications and pages tested for throughput and latency performance of Passe.

Amazon cites 300 ms as their desired 99.9% latency for requests [31]. In comparison, Passe’s overhead is not an excessive burden.

To characterize the throughput overhead of Passe, we used JMeter configured with 40 simultaneous users. We ran Gunicorn with 8 worker processes (one worker per CPU core). When running Django, each of these workers ran a separate Python interpreter with a separate copy of the Django application. When running Passe, each worker ran a separate Python interpreter connected to separate instances of Passe.
Throughput results (Figure 2.12) show that the cost of Passe may vary greatly based on the application. For the simple benchmark, which requires little I/O and no database operations, Passe reduces throughput by 37%. However, for actual applications that often require more I/O and query the database, such as the forum or CMS applications, we find Passe reduces throughput by about 25%.

### 2.8.2 Memory Overhead

In addition to latency overhead, Passe adds memory overhead, as each view requires a separate OS process. To characterize this overhead, we measured the total memory consumption of Passe while running 8 workers for our benchmark applications (Figure 2.13). While this memory overhead is large, it does not increase significantly under load. Rather, the memory overhead corresponds to the number of views in the application, each of which causes 8 separate processes to be spawned. These forked processes unfortunately do not make efficient usage of copy-on-write memory (CPython’s reference counting garbage collector requires that most pages copy), but
versions of Passe built for other languages or web frameworks may not suffer as much from forking.

In order to understand the behavior of this relationship, we varied the number of views in our simple application. Figure 2.14 shows the linear correspondence between the number of views and the memory consumption. Based on these measurements,
modern servers with a fairly modest 16 GB of RAM would be able to run applications with hundreds of views. In comparison, the most complicated application we tested had only 47 views. While on dedicated servers, memory consumption may not be prohibitive, in memory-constrained environments (e.g., hosted virtual machines), this cost may be excessive. In these cases, a memory optimized version of Passe could make better use of copy-on-write memory and other memory saving techniques to reduce this overhead.

2.9 Discussion and Possible Extensions to Passe

2.9.1 Interaction with Application Caching

![Diagram of application cache interaction](image)

Figure 2.15: Two typical deployments of an application cache. In this scenario, a proxy intermediates on GET requests to Passe. If the URL is cached, the proxy returns the data without communicating with Passe. Otherwise, Passe handles the request and the proxy caches the response. The second potential use of the cache is in the view, where a developer may use a cache as a look-aside cache to store temporary information for view code, or speed up accesses to the backing store.

As discussed in Chapter 1, and as we will see in Chapter 3, application caches are often deployed to drastically improve the performance of a web application. Passe’s threat model, however, impacts a cache’s deployment, even if the cache is itself within a trusted domain. For example, an attacker that has taken control over a view could write phony data to the cache or use the cache to read sensitive data. Of course,
Passe’s AppArmor protections prevent views from interacting with a cache as-is, but reintroducing caching functionality to an application would require mitigation of this channel of attack. Our model will examine two typical cache deployments, a whole-page cache (sometimes called a proxy cache) and a look-aside cache (see Figure 2.15).

A whole-page cache can be incorporated into Passe’s design almost directly. If the whole-page cache is managed by a trusted proxy process, then protecting this cache is simple. This is the case if the cache is deployed using Django’s whole page caching middleware, or if the proxy cache (e.g., nginx or Vanquish) are within an application developer’s trust domain. As long as the cache is only accessible to the trusted process, a compromised view will only be able to store bad data in entries associated with its own view. This is because the trusted process will store any pages returned from a view at a key corresponding to URLs handled by that view. Passe’s AppArmor policies will prevent views from otherwise communicating to the cache directly. A whole-page cache would help to mitigate the performance overheads experienced by Passe—when a request results in a hit, the cache could service the request without invoking any Passe code whatsoever.

A look-aside cache, however, poses a larger problem. A relatively simple approach to securing a look-aside cache with Passe could introduce a trusted intermediate process between the view server and the cache. This intermediate process would merely ensure than any key operations were prefixed with a view-server specific key. So, for example, a PUT on key FOO would be rewritten to a PUT on key VIEW-X-FOO. This would ensure that views interact with isolated portions of the cache’s keyspace. An attacker’s view could spam the cache with writes, essentially denying other views ability to interact with the cache (or attempt to use cache evictions as a side-channel), but Passe does not try to mitigate these kinds of attacks. Our prototype does not currently allow for application interactions with a cache, and our AppArmor protections prevent views from attempting to interact directly with a cache.
2.9.2 Applicability to Other Frameworks

While much of Passe’s prototype implementation is concerned with Django-specific modifications, Passe’s architecture is directly applicable to other web frameworks as well. For example, in Ruby on Rails, the dispatcher would make routing decisions based on Rails’ routes and views would be separated along Rails’ ActionControllers and Views. However, because some frameworks do not include standard authentication libraries, Passe would need to provide a new third party authentication library, or support some of the most common ones.

2.9.3 Policy File Language and Query-level Constraints

The policy files generated by Passe make assertions based on database queries. However, an application developer would not typically have to deal with raw SQL queries in an unmodified web framework. While developers concerned with performance already monitor SQL statements generated by their applications, the inspection of SQL queries required by Passe does place an additional burden on developers. A developer friendly extension to Passe could ease this process of reading and modifying the policy file by associating the Python statements which generated queries with the queries in the policy file.

2.9.4 Micro-rebooting

In Passe, view servers are not responsible for storing session state, pooling connections to the database, or caching data. As such, they do not store any state between requests. Thus, they could be rebooted from read-only application source code without affecting requests. Because so little state would need to be reinitialized—primarily, just their network sockets to the Passe dispatcher and database proxy—processes
could be quickly rebooted. This mechanism, similar to the micro-rebooting of VM components in XOAR [25], could mitigate the efficacy of remote code exploits.

While attackers could attempt to continuously re-exploit a view, such frequent suspicious activity would be likely to raise alarms if traffic is properly logged and monitored.

2.10 Related Work for Application Security

Intrusion Detection Systems   Our approach is most related to previous work in intrusion detection. Like intrusion detection systems, Passe uses dynamic analysis of execution to “learn” the normal or intended behavior of the application. Some work, such as DFAD [12] or SwitchBlade [40], has similarly used taint tracking to check whether a detected deviation in program behavior was the result of a code injection attack. This does not address attacks where code injection was not the root cause. Other intrusion detection work, such as Swaddler [29], SENTINEL [59] and DoubleGuard [56], has analyzed internal program state to infer potential invariants using Extended Finite State Machine modeling.

Passe differs from this work in two major ways. First, Passe actively splits applications into sandboxed components, allowing Passe to more easily infer constraints and to support more restrictive constraints than could otherwise be achieved. Second, Passe’s enforcement mechanisms operate without instrumenting application code or requiring a stateful database proxy. This prevents arbitrary code exploits from defeating the system and allows the proxy to be implemented in a scale-out architecture.

AutoISES [87] attempts to infer relationships between security checks and data accesses. In general, Passe cannot know which queries are “permission checks,” and so must make inferences about the relationships between queries.
Automated Vulnerability Detection  Some work in vulnerability detection has used a similar inference model to find potential errors in application code [35]. Several systems for detecting web vulnerabilities use program analysis to find bugs which can be readily identified once they are found (e.g., application crashes, malformed HTML, or forced browsing) [8,13,85]. For finding general data-flow violations which are harder to characterize, Passe cannot use the same kind of analysis.

Other work attempting to detect data-flow vulnerabilities has used a similar approach to Passe. For example, in [38], “normal” usage of the application is analyzed dynamically. In [67], taint tracking is used to identify cases in which user input has not been properly sanitized. Such work is focused on finding bugs rather than developing policies for a secure runtime, as in Passe. Thus, many of these projects’ mechanisms cannot be applied to Passe’s setting.

Decentralized Information Flow Control  Passe differs significantly from traditional DIFC systems [21,34,54,64,65,97], as Passe learns application policy during an analysis phase, while DIFC systems require developers or users to explicitly annotate data or code. Because DIFC systems require reasoning about information labels, application code may still be vulnerable to aberrant behavior. This is true even for automatic instrumentation systems such as SWIM [48], which still requires developer-supplied policies. Hails [44] applies the DIFC model to web applications while using a shared data store. Hails requires applications to be written in a safe subset of Haskell. Hails’ policies provide stronger guarantees than Passe, but require explicit policy specification.

XSS Protection  There has been significant work in preventing XSS attacks. Much of this work has focused on filtering and sanitizing user inputs. Passe addresses a stronger class of threats, in which the attack has compromised part of the application code. Other work allows applications to specify exactly which scripts should be
executed and in what form [66,89], or focuses on using the proposed CSP standard [26] to separate trusted script sources from the data of the HTTP response [33]. Other client side solutions use taint tracking or fine-grained policies to limit the threat of XSS attacks [62,92] XSS-Guard learns the set of scripts a web application typically constructs, and blocks unrecognized scripts [14]. While these approaches may work in Passe’s setting, the approach we chose reuses the view-level isolation model from the rest of our system’s protections. This allows us to unify isolation at the server with isolation at the client.

**Other Approaches to Web Security** Resin [94], a system which uses explicit policies to specify allowable data flows, can provide many of the same properties as Passe. However, because Resin relies on data-flow tracking in the application during runtime, it is susceptible to remote code exploits.

Systems such as Diesel [39] and OKWS [55] provide web frameworks strongly rooted in the principle of least privilege. Passe provides much richer constraints and does not require explicit separation from the developer. SELinks [86] supports enforcing explicit security policies near the database. Unlike Passe, policies are compiled into user-defined functions at the database.

### 2.11 Passe Summary

The Passe system demonstrates how to provide security guarantees for applications using a shared data store. Passe decomposes applications into isolated views which execute in sandboxed environments. Passe enforces the integrity of data queries by using cryptographic tokens to preserve learned data and control-flow dependencies. In doing so, Passe infers and enforces security policies without requiring developers to specify them explicitly (and sometimes erroneously).
Our Passe prototype is capable of executing unmodified Django applications. We test Passe on eleven off-the-shelf applications, detail some of its inferred constraints, demonstrate several examples of security vulnerabilities it prevents, and show that it adds little performance overhead.
Chapter 3

Flexible Application Caching with Hyperbolic Caching

3.1 Caching Overview

Web applications and services aggressively cache data originating from a backing store in order to reduce both access latency and backend load. The wide adoption of Memcached [42] and Redis [76] (key-value caching), Guava [46] (local object caching), and Varnish [90] (front-end HTTP caching) speak to this demand, as does their point-and-click availability on cloud platforms like Heroku via MemCachier [61], EC2 via ElastiCache [5], and Azure via Azure Redis Cache [10].

Caching performance is determined by the workload and the caching algorithm, i.e., the strategy for prioritizing items for eviction when the cache is full. All of the above services employ an inflexible caching algorithm, e.g., LRU. But the needs of each application vary, and significant performance gains can be achieved by tailoring the caching strategy to the application: e.g., incorporating item frequency, cost of fetching, or other factors [11,81]. In the limit, function-based strategies [2,93] devise general functions that combine several of these factors.
All of these strategies are fundamentally limited, however, because they rely on data structures (typically priority queues) to track the ordering of cached items. In particular, an item’s priority is only changed when it is accessed. However, does cache eviction need to be tied to data structures? Caches like Redis already eschew ordering data structures to save memory [77]. They rely instead on random sampling to evict items: a small number of items are sampled from the cache, their priorities are evaluated (based on per-item metadata), and the item with lowest priority is evicted. This approach evicts the approximately lowest-priority item [74]. Now, we ask, can this lack of an ordering data structure allow us to build a framework with vast flexibility in how it is used? Indeed, we show that the combination of random sampling and lazy evaluation enables us to evolve item priorities arbitrarily, allowing us to freely explore the design space of priority functions. We find that this approach outperforms many traditional and even domain-optimized algorithms, yet neither Redis nor existing algorithms exploit it.

Armed with this flexibility, we systematically design a new caching algorithm for modern web applications, called hyperbolic caching (Section 3.2). We begin with a simple theoretical model for web workloads that leads to an optimal solution based on frequency. A key intuition behind our approach is that caches can scalably measure item frequency only while items are in the cache. Thus, we overcome the drawbacks of prior frequency-based algorithms by incorporating the time an item spends in the cache. Yet this deceptively simple modification is already infeasible using an ordering data structure, as pervasively employed today, because item priorities decay at variable rates and are continuously being reordered. Yet with hyperbolic caching, we can easily customize it for different scenarios by adding extensions to the priority function, such as for item cost, expiration time, and windowing (Section 3.3). We also introduce the notion of cost classes to manage groups of related items, e.g., items materialized by the same database query. Classes enable us both to more accurately
measure item’s miss cost (to overcome the stochastic nature of individual requests) and to adjust the priorities of many items at once (e.g., in response to the database becoming overloaded).

A quick survey of existing algorithms shows that they fall short of this flexibility in different ways. Recency-based algorithms like LRU use time-of-access to order items, which is difficult to extend: for example, incorporating costs requires a completely new design (e.g., GreedyDual [95]). Frequency-based algorithms like LFU are easier to modify, but any non-local change to item priorities—e.g., changing the cost of multiple items—causes expensive churn in the underlying data structure. Some algorithms, such as those based on marking [41], maintain only a partial ordering, but the coarse resolution makes it harder to incorporate new factors. Several theoretical studies [2, 81] formulate caching as an optimization problem unconstrained by any data structure, but their solutions are approximated by online heuristics that, once again, rely on data structures.

We design a hyperbolic caching variant for several different production systems from leading cloud providers (Section 3.3), and evaluate them on real traces from those systems. We implement hyperbolic caching in Redis and the Django web framework [32], supporting both per-item costs and cost classes (Section 3.4). Overall (Section 3.5), we find that hyperbolic caching reduces misses substantially over competitive baselines tailored to the application, and improves end-to-end system throughput by 5-10%. And this not insignificant improvement arises from simply changing the caching algorithms employed by existing systems—our modification to Redis was 380 lines of code—with no other changes needed.

To summarize, we make the following contributions:

1. We systematically design a new caching algorithm for modern web applications, hyperbolic caching, that prioritizes items in a radically different way.
2. We define extensions for incorporating item cost and expiration time, among others, and use them to customize hyperbolic caching to three production systems.

3. We introduce the notion of cost classes to manage groups of related items effectively.

4. We implement hyperbolic caching in Redis and Django and demonstrate performance improvements for several applications.

Although we only evaluate medium-to-large web applications, we believe hyperbolic caching can improve hyper-scale applications like Facebook, where working sets are still too large to fit in the cache [9, 88].

### 3.2 Hyperbolic Caching

We first describe the caching framework required by hyperbolic caching (Section 3.2.1). Then, we motivate a simple theoretical model for web workloads and show that a classical frequency approach is optimal in this model (Section 3.2.2). By solving a fundamental challenge of frequency-based caching (Section 3.2.3), we arrive at hyperbolic caching (Section 3.2.4).

#### 3.2.1 Framework

We assume a caching service that supports a standard get/put interface. We make two changes to the implementation of this interface. First, we store a small amount of metadata per cached item $i$ (e.g., total number of accesses) and update it during accesses; this is done by the on_get and on_put methods in Figure 3.1. We do not (currently) store metadata for non-cached items. Second, we remove any data structure code that was previously used to order the items. We replace this with a priority function $p(i)$ that maps item $i$’s metadata to a real number; thus $p$ imposes a total ordering on the items. To evict an item, we randomly sample $S$ items from the cache.
def evict_which():
    sampled_items = random_sample(S)
    return argmin(p(i) for i in sampled_items)

def on_put(item):
    item.accessed = 1
    item.ins_time = timenow()
    add_to_sampler(item)

def on_get(item):
    item.accessed += 1

def p(item):
    in_cache = timenow - item.ins_time
    return item.accessed / (in_cache)

Figure 3.1: Pseudocode for hyperbolic caching in our framework.

and evict the item \( i \) with lowest priority \( p(i) \), as implemented by \texttt{evict\_which}. This approximates the lowest-priority item \cite{74}; we evaluate its accuracy in Section 3.5.3.

The above framework is readily supported by Redis, which already avoids ordering data structures and uses random sampling for eviction. The use of metadata and a priority function is standard in the literature and referred to as “function-based” caching \cite{11}. What is different about our framework is \textit{when} this function is evaluated. Prior schemes \cite{2,81,93} evaluate the function on each \texttt{get/put} and use the result to (re)insert the item into a data structure, freezing its priority until subsequent accesses. Our framework uses \textit{lazy evaluation} and no data structure: an item’s priority is only evaluated when it is considered for eviction, and it can evolve arbitrarily before that point without any impact on performance.

3.2.2 Model and Frequency-based Optimality

In many workloads, the requests follow an item popularity distribution and the time between requests for the same item are nearly independent \cite{18}. Absent real data,
most systems papers analyze such distributions (e.g., Zipfian [36, 98]), and model
dynamism as gradual shifts between static distributions. Motivated by this, we model
requests as a sequence of static distributions \( \langle D_1, D_2, \ldots \rangle \) over a universe of items,
where requests are drawn independently from \( D_1 \) for some period of time, then from
\( D_2 \), and so on. Our measure of cost is the miss rate, which is widely used in practice.
The model can be refined by constraining the transitions \((D_i, D_{i+1})\), but even if we
assume they are instantaneous, we can prove some useful facts. We summarize these
below.

Within a distribution \( D_i \), a simple application of the law of large numbers shows
that the optimal strategy for a cache of size \( k \) is to cache the \( k \) most popular items.
This is closely approximated by the least-frequently-used (LFU) algorithm: a typical
implementation assigns priority \( n_i / T \) to item \( i \), where \( n_i \) is number of hits to \( i \) and
\( T = \sum n_i \) is the sum over all cached items. Whereas LFU approximates the optimal
strategy, one can prove that LRU suffers a gap. This is in contrast to the traditional
competitive analysis model which assumes a worst-case request sequence and use
total misses as the cost [82], for which LRU is optimal. This model has been widely
criticized (and improved upon) for being pessimistic and unrealistic [4,17,52,53,96].
Our model is reminiscent of older work (e.g., [43]) that studied independent draws
from a distribution but, again, used total misses as the cost.

To validate our theoretical results, we use a static Zipfian popularity distribution
and compare the miss rates of LRU and LFU to the optimal strategy, which has perfect
knowledge of every item’s popularity (Figure 3.2).\(^1\) Until the cache size increases to
hold most of the universe of items, LRU has a 25-35% higher miss rate than optimal.
LFU fares considerably better, but is far from perfect. We address the drawbacks of

\(^1\)We present miss rate rather than hit rate curves because our focus is on the penalties at the backend.
Higher numbers indicate worse performance in most figures and the last datapoint is 0 because the
cache is big enough to never suffer a miss.
Figure 3.2: Miss rate curves\(^1\) of LFU and LRU compared to a strategy with perfect frequency knowledge. Items are sampled with Zipfian popularity (\(\alpha \approx 1\)) from a universe of \(10^5\) items. The simulated cache is configured to hold a fixed number of objects (rather than simulating size in bytes).

### 3.2.3 Problems with Frequency

Even if requests are drawn from a stable distribution, there will be irregularities in practice that cause well-known problems for frequency-based algorithms:

**New items die.** When an item is inserted into the cache, the algorithm does not have a good measure of it’s popularity. In LFU, a new item gets a frequency count of 1, and may not have enough time to build up its count to survive in the cache. In the worst case, it could be repeatedly inserted and evicted despite being requested frequently. This problem can be mitigated by storing metadata for non-cached items (e.g., [60]), but at the cost of additional memory that is worst-case linear in the universe size.

**Old items persist.** When items’ relative popularities shift—e.g., moving from \(D_i\) to \(D_{i+1}\) in our model—a frequency approach may take time to correct its frequency
estimates. This results in older items persisting in the cache for longer than their current popularity warrants.

As a concrete example, consider a new item with 1 access and an older item with 2 accesses. Initially, the new item may be the better choice to cache, but if time passes without an additional access, our knowledge of the old item is more reliable.

### 3.2.4 Hyperbolic Caching

We solve the above problems by incorporating a *per-item* notion of time. Intuitively, we want to compensate for the fact that caches can only measure the frequency of an item *while it is in the cache*. Traditional LFU does not account for this, and thus overly punishes new items.

In our approach, an item’s priority is an estimate of its frequency since it entered the cache:

\[ p_i = \frac{n_i}{t_i} \tag{3.1} \]

where \( n_i \) is the request count for \( i \) since it entered the cache and \( t_i \) is the time since it entered the cache. This state is erased when \( i \) is evicted. Figure 3.1 provides pseudocode for this policy, which we call *hyperbolic caching*.

Hyperbolic caching allows a new item’s priority to converge to its true popularity from an initially high estimate. This initial estimate gives the item temporary immunity (similar to LRU), while allowing the algorithm to improve its estimate of the item’s popularity. Over time, the priority of each item drops along a hyperbolic curve. Since each curve is unique, the ordering of the items is continuously changing. This property is uniquely enabled by our framework (lazy evaluation, random sampling), and would be costly to implement with a data structure.\(^2\)

\(^2\)The basic hyperbolic function in Eq. 3.1 can be tracked by a kinetic heap [51], but this is a non-standard structure with \( O(\log^2 n) \) update time, and it ceases to work if the extensions from Section 3.3 are added.
Figure 3.3: LFU miss rate compared to hyperbolic caching (HC) on a dynamic Zipfian workload ($\alpha \approx 1$), where new items are introduced into the distribution every 100 requests.

The strengths of hyperbolic caching over LFU are readily apparent in workloads that slowly introduce new items into the request pool. Figure 3.3 shows that LFU has a significantly higher miss rate on a workload that introduces new items every 100 requests whose popularities are in the top 10% of a Zipfian distribution. This workload is artificial and much more dynamic than we would expect in practice, but serves to illustrate the difference.

Another way to solve the same problem is to multiplicatively degrade item priorities (e.g., LRFU [57]) or periodically reset them. Both of these are forms of windowing, which best addresses the problem of old items persisting, not the problem of new items dying. We compare hyperbolic caching to these approaches in Section 3.3.4.

### 3.3 Customizing Hyperbolic Caching

Our framework allows us to build on the basic hyperbolic caching scheme by adding extensions to the priority function and storing metadata needed by those extensions. This is similar to the way function-based policies build on schemes like LRU and LFU [2,81,93], but in our case the extensions can freely perturb item priorities with-
out affecting efficiency (beyond the overhead of evaluating the function). Which extensions to use and how to combine them are important questions that depend on the application. Here, we describe several extensions that have benefited our production applications (cost, expiration time) and our understanding of hyperbolic caching’s performance (windowing, initial priority estimates).

### 3.3.1 Cost-aware Caching

In cost-aware caching, all items have an associated cost that reflects the penalty for a miss on the item. The goal is to minimize the total cost of all misses. Cost awareness is particularly relevant in web applications, because unlike traditional OS uses of caching (fixed-size CPU instruction lines, disk blocks, etc.), the cost of fetching different items can vary greatly: items vary in size, can originate from different backing systems or stores, or can be the materialized result of complex database joins.

Much of the prior work on cost-aware caching focuses on adapting recency-based strategies to cost settings (e.g., GreedyDual [20]). This typically requires a new design, because recency-based strategies like LRU-K [69] and ARC [60] use implicit priorities (e.g., position in a linked list) and metrics like time-of-access, which are difficult to augment with cost. In contrast, frequency-based approaches like hyperbolic caching use explicit priorities that can naturally be multiplied by a cost: 

$$p'_i = c_ip_i,$$

where $c_i$ is the cost of fetching item $i$ and $p_i$ is the original (cost-oblivious) priority of $i$. Note that $p_i$ may include other extensions from later sections.

The cost of an item needs to be supplied to the caching algorithm by the application. It can take many forms. For example, if the goal is to limit load on a backing database [58], the cost could be request latency. If the goal is to optimize the hit rate per byte of cache space used, the cost could be item size [20].
3.3.2 Cost Classes

In many applications, the costs of items are related to one another. For example, some items may be created by database joins, while others are the result of simple indexed lookups. Rather than measuring the cost of each item individually, we can associate items with a cost class and measure the performance of each class. We store a reference to the class in each item’s metadata.

Cost classes have two main advantages. Consider the example of request latency to a backend database. If costs are maintained per item, latencies must be measured for each insertion into the cache. Since these measurements are stochastic, some requests will experience longer delays than others and thus be treated as more costly by the cache, even though the higher cost has nothing to do with the item itself. What’s more, the higher costs will keep these items in the cache longer, preventing further updates because costs are only measured when a miss occurs. By using cost classes, we can aggregate latency measurements across all items of a class (e.g., in a weighted moving average), resulting in a less noisy estimate of cost.

The second advantage of cost classes comes from longer-term changes to costs. In scenarios where a replica failure or workload change affects the cost of fetching a whole class of items, existing approaches would only update the individual costs after the items have been evicted, one by one. However, when using cost classes, a change to a class’s cost is immediately applied to both newly cached items and items already in the cache.

In both cases above, a single update to a cost class changes the priorities of many items at once (potentially dramatically). Our framework supports this with little additional overhead because 1) items store a reference to the class information, and 2) priorities are lazily evaluated. In contrast, integrating cost classes into existing caching schemes is prohibitively costly because it incurs widespread churn in the data structures they depend on.
Interestingly, some production systems already employ cost classes implicitly, via more inflexible and inelastic means. For example at Facebook, the Memcached service is split among a variety of pools, such that keys that are accessed frequently but for which a miss is cheap do not interfere with infrequently accessed keys for which a miss is prohibitively expensive [68]. However, this scheme is much more management intensive and requires tuning pool sizes, and it does not automatically adapt to changes in request frequencies or item costs.

**Application to Django.** Django is a Python framework for web applications that includes a variety of libraries and middleware components. One such component is the Django caching middleware, which adds whole-page caching. We modify this middleware to support cost awareness: on a miss, the CPU time and database time required to render a page serves as the cost. In Django, page requests are dispatched to “view” functions based on the URL. We associate a cost class with each view function, and map individual pages to their view function’s class.

### 3.3.3 Expiration-aware Caching

Many applications need to ensure that the content conveyed to end users is not stale. Developers achieve this by specifying an *expiration time* for each item, which tells the caching system how long the item remains valid. While many systems support this feature, it is typically handled by an auxiliary process that has no connection to the caching algorithm (apart from evicting already-expired items). But incorporating expiration into caching decisions makes intuitive sense: if an item is soon to expire, it is unlikely to be requested before expiring, so evicting it is less costly than evicting a similarly popular item that expires later (or not at all).

To add expiration awareness to hyperbolic caching, we need to strike a balance between the original priority of an item and the time before it expires. Rather than
evict the item least likely to be requested next, we want to evict the item most likely to be requested the least number of times over its lifetime. This can be naturally captured by multiplying item $i$’s priority by the time remaining until expiry, or $\max((t_{exp, i} - t_{cur}), 0)$. However, this scheme equally prioritizes requests far into the future and those closer to the present, which is unideal because estimates about the future are less likely to be accurate (e.g., the item’s popularity may change). Therefore, instead of equally weighting all requests over time, we use a weighting function that discounts the value of future requests:

$$p_i' = p_i \cdot (1 - e^{-\lambda \cdot \max((t_{exp, i} - t_{cur}), 0)})$$

where $p_i$ is the original (expiration-unaware) priority of item $i$ and $\lambda$ is a parameter controlling how quickly to degrade the value of future requests. As an item’s time until expiration decreases, this weighting function sharply approaches zero. Thus the function continually reweights item priorities, which is uniquely possible in our frame-
work: existing approaches can only account for expiration time once, on insertion into a data structure.

Figure 3.4 shows simulation results from a synthetic workload where items are either set to expire after 100 requests or set to never expire. When the cache size is $\geq 10^4$, expiration-aware hyperbolic caching begins to dramatically outperform standard hyperbolic caching. When the cache is big enough to contain the whole workload, however, the difference collapses back to 0%.

3.3.4 Windowing

Windowing is often used in frequency-based caching to adapt to dynamic workloads and address the problem of “old items persisting”. The idea is to forget requests older than a fixed time window from the present. Hyperbolic caching naturally achieves the benefits of windowing, but we investigate it for two reasons. First, one can show that hyperbolic caching, unlike LRU, is not optimal in the traditional competitive analysis model [82], but it can be made optimal if windowing is used (proof omitted). Second, windowing represents alternative solutions, such as resetting or multiplicatively degrading frequency estimates (e.g., LRFU [57]), and so this serves as an informative comparison.

We simulate windowing using an idealized (but completely inefficient) scheme that tracks every request and forgets those older than the window. This serves to upper bound the potential gains of windowing. Figure 3.5 shows the performance of LFU and hyperbolic caching on a dynamic Zipfian workload, with and without windowing. For hyperbolic caching, windowing provides limited benefits: 5–10% reduction in misses on small cache sizes; LFU benefits more but again on small cache sizes. The problem is that windowing discards measurements that help the cache estimate item popularity. Even in dynamic workloads, we find that large-sized caches can accommodate newly popular items, so performance depends more on the ability
Figure 3.5: Adding perfect windowing to hyperbolic caching and LFU on a dynamic Zipfian workload ($\alpha \approx 1$). Each curve is compared to the algorithm’s non-windowed performance (given in the table). The window size is fixed at $10^4$ requests. Every 100 requests, an item is promoted to the top of the distribution.

to differentiate at the long tail of old items. Fortunately, hyperbolic caching’s measure of time in the cache achieves some of the benefits of windowing; it outperforms even recency-based approaches on many of the highly dynamic workloads we evaluated.

### 3.3.5 Initial Priorities

Hyperbolic caching protects newly cached items by giving them an initial priority that tends to be an overestimate: for example, an item with true popularity of 1%—placing
Figure 3.6: Miss rate of hyperbolic caching with initial priority estimates \((\beta = 0.1)\) on a Zipfian workload \((\alpha \approx 1)\), compared to a baseline (HC) using an initial estimate of 1. Miss rates of baseline are provided in the table.

We found that by tracking the priority of recently evicted items, we can use this priority to seed the priorities for items entering the cache. This performs well because evicted items will tend to come from the tail of the popularity distribution and items in the tail will tend to have similar priorities (this same intuition explains the effectiveness of random sampling). Thus, we set a new item’s initial priority to a mixture of its original priority \((p_i)\) and the last evicted item’s priority \((p_e)\).

\[
p'_i = \beta p_i + (1 - \beta) p_e
\]

Solving this for \(n_i\) in Equation 3.1 gives us the initial request count to use, after which the extension can be discarded. \(\beta\) requires some tuning: we found that \(\beta = 0.1\) works well on many different workloads; for example, on a Zipfian workload \((\alpha \approx 1)\) it reduced the miss rate by between 1% and 10% over hyperbolic caching for all cache sizes (see Figure 3.6).
3.4 Implementation

Our evaluation uses both simulation and a prototype implementation. For the simulations, we developed a Python application that generates miss rate curves for different caching strategies and workloads. For our prototype, we implemented hyperbolic caching in Redis and developed Django middleware that uses the modified Redis. Our code is open-source [49].

Redis. We modified Redis (forked at 3.0.3) to use the hyperbolic caching framework. This was straightforward because Redis already uses random sampling for eviction. We included support for the following extensions: per-item costs (also used for size awareness), cost classes tracked with an exponentially weighted moving average, and initial priorities. Excluding diagnostic code, this required 380 lines of C code, of which 130 were for setting/getting costs.

We store the following metadata per item, using double-precision fields: item cost, request count, and time of entry (from Eq. 3.1 and Section 3.3.1). Our prototype achieved similar miss rates to our simulations, suggesting this precision is adequate. This is two doubles of overhead per item compared to LRU. Exploring the trade-offs of reduced resolution in these fields is left for future work.

Django caching middleware. Django is a framework for developing Python web applications. It includes support for middleware classes that enable various functionality, such as the Django whole-page caching middleware. This middleware interposes on requests, checking a backend cache to see whether a page is cached, and if so, the content is returned to the client. Otherwise, page processing continues as usual, except that the rendered page is cached before returning to the client. We added middleware to track cost information for web pages; we measure cost as the CPU time between the initial miss for a page and the subsequent SET operation, plus the
total time for database queries. This avoids time lost due to processor scheduling. We subclassed the Django Redis caching interface to convey cost information to our Redis implementation. The interface supports caching a page with/without costs, and optionally specifying a cost class for the former. Cost classes are associated with the particular Django “view” function that renders the page. In total, this was implemented in 127 lines of Python code.

3.5 Evaluation

Our evaluation explores the following questions:

1. How does hyperbolic caching compare to current caching techniques in terms of miss rate?

2. Does our implementation of hyperbolic caching in Redis improve the throughput of web applications?

3. What effect does sample size have on the accuracy and performance of our eviction strategy?

We use real application traces (Section 3.5.1) and synthetic workloads designed to emulate realistic scenarios (Section 3.5.2). We evaluate these questions using simulations as well as deployments of Django and NodeJS, using our prototype of hyperbolic caching on Redis. To drive our tests, our applications run on Ubuntu 14.04 servers located on a single rack with Intel Xeon E5620 2.40GHz CPUs. Applications use PostgreSQL 9.3 as their backing database.

Methodology. For the majority of our standard workloads, we use a Zipfian request distribution with $\alpha \approx 1$. This is the same parameterization as many well-studied benchmarks (e.g., YCSB [27]), though some benchmarks like Facebook’s linkbench
use a heavier-tailed $\alpha = 0.9$ \[7\]. When measuring miss rates, we tally misses after the first cache eviction. For workloads with associated item costs, misses are scaled by cost. For real traces, we run the tests exactly as prescribed, while for workloads based on popularity distributions, we generate enough requests to measure the steady state performance. When choosing a cache size to compare performance amongst algorithms, we use the size given by the trace, or if not given, we choose sizes corresponding to high and middle range hit rates (90% and 70%). For our random sampling, unless otherwise noted, we sample 64 items.

### 3.5.1 Real-world Workloads

We evaluate real applications in two ways. When lacking access to the actual application code or deployment setting, we evaluate the performance through simulation. For other applications, we measure the performance using our prototype implementation of Django caching paired with Redis.

**Memcachier Applications**

Memcachier \[61\] is a production cloud-based caching service built on Memcache. We examined the caching behavior of a set of applications using Memcachier. We used costs to incorporate request size into the eviction decision, i.e., set $c_i = 1/s_i$ where $s_i$ is the size of item $i$.

To evaluate the Memcachier applications, we processed a trace of GET and SET requests spanning hundreds of applications, \(^3\) using the amount of memory allocated by each application as the simulated cache size. We focused our attention on the 16 applications with over 10k requests each whose allocation could not fit all of the requested objects (many applications allocated enough memory to avoid any evictions). We measured the miss rates of plain HC and LRU, and then used the object sizes to

---

\(^3\)While memcached supports other kinds of requests, we do not handle these in our simulation.
evaluate our size-aware extension, HC-Size, and the GD-Size [20] algorithm. Fig. 3.7 show the performance of the algorithms over a single execution of each application’s trace.

In our evaluation, HC outperforms LRU in many applications, and HC-Size drastically outperforms LRU. While GD-Size is competitive with HC-Size, our framework allows for the implementation of HC-Size with only two lines of code, whereas implementing GD-Size from LRU would require an entirely new data structure [20].

**Decision Service**

The Decision Service [1,63] is a machine learning system for optimizing decisions that has been deployed in MSN to personalize the news articles shown to users. Given a user request \( \text{id} \), a particular article is chosen to appear in the top slot (the decision), and the outcome (click or no click) is determined. Since outcomes may occur after a
substantial delay, a “join service” is used to combine the decision \((id, decision)\) with the outcome \((id, click?)\). This service is implemented with a cache. Click events are only valid if they occur within a window of time after the decision (e.g., 10 min in the case of MSN): this defines an expiration time for each item.

In this workload, because items have the same expiration time and are accessed only once after insertion (to join the reward information), recency is roughly equal to time-until-expiration. Therefore, LFU and HC perform poorly in comparison to a recency strategy (Fig. 3.1). However, our expiration-aware extension allows HC-Expire to perform just as well as the recency strategies.

### Viral Search

The Viral Search [45, 91] application is a Microsoft website that displays viral stories from the Twitter social network. Virality is measured by analyzing the diffusion tree of the story as it is shared through the network; these trees are stored as edges in a backend database. To display a story’s tree, the website must fetch the edges from the database, construct the tree, and lay it out for display. The final trees are cached and the cost \(c_i\) of tree \(i\) is set to the time required to construct and lay out the tree.

<table>
<thead>
<tr>
<th></th>
<th>Decision Service</th>
<th>Viral Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>1k</td>
<td>35k</td>
</tr>
<tr>
<td>Algo.</td>
<td>Miss Rate ((\Delta%))</td>
<td>Miss Rate ((\Delta%))</td>
</tr>
<tr>
<td>HC</td>
<td>0.60 (+0%)</td>
<td>0.17 (+0%)</td>
</tr>
<tr>
<td>HC-Expire</td>
<td>0.55 (-8%)</td>
<td>—</td>
</tr>
<tr>
<td>LRU/GD</td>
<td>0.55 (-8%)</td>
<td>0.18 (+6%)</td>
</tr>
<tr>
<td>ARC</td>
<td>0.55 (-8%)</td>
<td>0.22 (+29%)</td>
</tr>
<tr>
<td>LFU</td>
<td>0.99 (+65%)</td>
<td>0.16 (-6%)</td>
</tr>
</tbody>
</table>

Table 3.1: Miss rates on Decision Service and Viral Search apps.
Django Wiki Application

We evaluate our caching scheme on an open-source Django wiki app using our Django caching middleware. The caching middleware stores cached data using a configurable backend, for which we use either the default Redis or our modified version with hyperbolic caching.

The wiki database serves a full copy of articles on Wikipedia from Jan. 2008. We measured the throughput and miss rate of the application using a trace of Wikipedia article requests from Sept. 1, 2007 (Fig. 3.2). We see an improvement in both miss rate and throughput when using HC rather than default Redis. Note that because the pages are costly to render, even small improvements in miss rate increase the throughput of the application.

However, using HC-Cost reduces the system throughput compared to HC. This is because the time to render a page is similar across most pages, but has high variance: for one page, the mean time of fifty requests was 570ms with a deviation of 180ms.

To evaluate the caching behavior of this application, we generate a request distribution such that items are requested based on a popularity given by each item’s “virality score”. We measure performance over 10M requests.

Hyperbolic caching performs very well on this cost-aware workload, beating all algorithms except for LFU (Fig. 3.1), and suffering 6% fewer misses than GreedyDual.

Table 3.2: Performance of Django-Wiki application using HC Redis compared to default Redis. The cache is sized to 1GB and the workload is a 600k trace of Wikipedia article requests.
This leads a cost-aware strategy to incorrectly favor some pages over others. HC-Class alleviates this by reducing some of the variance, but it still incurs a higher miss rate and lower throughput than the cost-oblivious HC. For this application, using costs is counter-productive.

**ARC and SPC Traces**

We additionally simulate performance on traces from ARC [60] and SPC [84] (Fig. 3.8). The P1-4 traces are memory accesses from a workstation computer; S1 and WebSearch are from a server handling web searches; and the Financial workload is an OLTP system trace. Caches were sized according to the ARC paper, and these sizes were used for the SPC traces as well. These traces have very high miss rates on all eviction strategies. However, HC performs very well, outperforming LRU in every workload and underperforming ARC in the P1-4 traces only. Importantly, on workloads where LFU exhibits poor performance, HC remains competitive with ARC, demonstrating the effectiveness of our improvements over LFU.
Figure 3.9: Miss rates on synthetic workloads with 10M requests. Miss rates are compared to the performance of HC. For cost-aware strategies (GD1-GD3), misses are scaled by the cost of the missed item.

### 3.5.2 Synthetic Workloads

In this section, we simulate and compare the performance of HC to three popular strategies—ARC, LFU, and LRU—on synthetic workloads that reflect the demands of today’s caches. For cost-aware workloads, we extend LRU with GreedyDual, and we modify LFU by multiplying frequencies by cost. (ARC is not amenable to costs.)

For each synthetic workload, we evaluate the performance of each caching algorithm on two cache sizes, corresponding to a 90% and a 70% hit rate with hyperbolic caching (Fig. 3.9). Note that while we simulated relatively small key spaces, we evaluated our Redis prototype on larger key spaces and larger cache sizes, and found
Figure 3.10: Simulated miss rate of LRU compared to HC on Zipfian workloads of varying skew.

similar improvements in miss rate and overall system throughput. In general, these workloads suggest that HC can perform very well in a variety of scenarios.

The most striking improvement relative to ARC is on workloads GD1-3. These workloads have associated costs and are based on the workloads described in GDWheel [58]. Since ARC is a cost-oblivious strategy, it does poorly on these workloads. However, even in workloads without cost, our scheme is competitive with ARC.

While LFU performs competitively with HC on several workloads, it exhibits dramatically poor behavior on the dynamic workloads. The first of these, DynamicIntro, introduces a new item into the key space with a high popularity (top decile) every 100 requests. When the new items are introduced, LFU takes very long to phase old items out of the cache. As shown previously in Figure 3.2 (Section 3.2.2), LFU’s performance compared to HC degrades as the cache size grows. Only once the cache can contain nearly the whole dataset does LFU perform competitively. While LRU performs much better than LFU, it still performs worse than HC. The second dynamic workload, DynamicPromote, randomly selects an item from the key space and promotes its popularity to the cache’s top decile every 100 requests.
Figure 3.11: Throughput of NodeJS using Redis as a look-aside cache for PostgreSQL as the miss rate varies.

**Varying Skewness of Distribution.** To demonstrate that HC outperforms a traditional strategy on a wide range of popularity skews, we compared the simulated miss rate of HC to LRU as we swept the Zipfian skew parameter $\alpha$ (Fig. 3.10). For heavier tailed distributions (lower $\alpha$), HC outperforms LRU by over 10%, even as the cache can store about a tenth of the universe of items. For lighter tailed distributions, HC also outperforms LRU, but the performance gap closes more quickly as the cache size increases (as both cache algorithms will be able to easily store the most important items).

**Synthetic web application performance.** In order to understand how our improved miss rates affect end-to-end throughput in modern web servers, we configured a NodeJS web app to use a backing database with Redis as a look-aside cache. We drive HTTP GET requests to our web app from a client that draws from synthetic distributions. The NodeJS app parses the URL and returns the requested object. Objects are stored as random 32B strings in a table with object identifier as the primary key.

**Relating cache misses to throughput.** To understand the association between miss rate and throughput, we scaled the size of our Redis cache to measure system
throughput with different miss rates (Fig. 3.11). Miss rate has a direct impact on throughput even when many client requests can be handled concurrently. Misses not only cause slower responses from the backend (an effect which can be mitigated with asynchronous processing), but they also require additional processing on the web server—on a miss, the web app issues a failed GET request, a SQL SELECT, and then a PUT request on the cache. This adds a direct overhead to the throughput of the system.

**Zipfian distribution.** We measured the maximum throughput of our NodeJS server when servicing requests sampled from synthetic workloads with zipfian request distributions (Fig. 3.3.) Depending on the workload, hyperbolic caching outperforms Redis’s default caching algorithm (LRU approximated by random sampling) in miss

<table>
<thead>
<tr>
<th>Size (objs.)</th>
<th>Default Redis</th>
<th>HC Redis</th>
<th>Δ tput.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean miss tput. rate</td>
<td>Mean miss tput. rate</td>
<td></td>
</tr>
<tr>
<td>Zipfian ($\alpha \approx 1, N = 10^5$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39k</td>
<td>18.1 ± 0.22 0.11</td>
<td>20.2 ± 0.18 0.09</td>
<td>10.3%</td>
</tr>
<tr>
<td>3k</td>
<td>9.1 ± 0.09 0.38</td>
<td>10.5 ± 0.06 0.31</td>
<td>13.5%</td>
</tr>
<tr>
<td>Zipfian ($\alpha = 0.75, N = 10^6$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>125k</td>
<td>7.5 ± 0.06 0.55</td>
<td>7.7 ± 0.16 0.49</td>
<td>3.2%</td>
</tr>
<tr>
<td>70k</td>
<td>6.8 ± 0.06 0.64</td>
<td>7.3 ± 0.12 0.56</td>
<td>6.3%</td>
</tr>
<tr>
<td>Zipfian ($\alpha \approx 1, N = 10^6$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200k</td>
<td>14.6 ± 0.16 0.17</td>
<td>15.3 ± 0.13 0.16</td>
<td>4.4%</td>
</tr>
<tr>
<td>50k</td>
<td>11.2 ± 0.11 0.28</td>
<td>12.1 ± 0.20 0.24</td>
<td>7.1%</td>
</tr>
<tr>
<td>Dynamic Intro. ($N = 10^5$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42k</td>
<td>19.3 ± 0.17 0.10</td>
<td>20.6 ± 0.16 0.09</td>
<td>6.3%</td>
</tr>
<tr>
<td>5k</td>
<td>10.0 ± 0.15 0.33</td>
<td>11.3 ± 0.12 0.27</td>
<td>11.6%</td>
</tr>
</tbody>
</table>

Table 3.3: Miss rate and throughput of workloads running on NodeJS with a Redis cache. Each configuration was executed 10 times with workloads of 5M requests to objects of size 96B.
rates by 10-37%, and improves throughput by up to 14% on some workloads. In particular, for the standard Zipfian distribution ($\alpha \approx 1$), our strategy provides significant throughput benefits. While throughput differences of 5-10% on other workloads may be modest, they are not insignificant, and come with little implementation burden.

**Cost-aware caching.** To measure the potential throughput benefits of cost-aware caching, we wrote a NodeJS application that makes two types of queries to the backend: the first is a simple key lookup and the second is a join. The application measures the latency of backend operations and uses that as the item’s cost. For our experiment, we sized the cache to hold 30k objects, and we drive the application with 1M requests sampled from a Zipfian distribution ($\alpha \approx 1$). When using cost-oblivious HC, we measured a throughput of 5070 reqs/s and a miss rate of 0.11. When using HC-Cost, the miss rate was .17, which is 57% higher, but the throughput was 9425 reqs/s, an 85% improvement over HC. HC-Cost traded off miss rate for lower overall cost, increasing overall performance.

**Responding to backend load with classes.** To demonstrate how cost classes can be used to deal with backend load, we designed a NodeJS application which performs key lookups on one of two different PSQL servers. The application measures the latency of the backend operation and uses that as the cost in our Redis prototype. Additionally, it sets the class of each cached object to indicate which backend served the object. This way, HC-Class will use a per-class cost estimate (exponentially WMA) when deciding which items to evict, rather than per-item. We evaluate the application by driving it with requests and measuring throughput and tail latency (Fig. 3.12). Two minutes into our test, we generate load on one of the PSQL backends using the Unix script **stress**. When one backend is loaded, throughput decreases and tail latency increases. By using per-class costs, HC-Class quickly adjusts to one class of items being more costly. With per-item costs, however, HC-Cost is only able to
update the costs of items when they are (re)inserted. As a result, HC-Cost needs more time to settle to steady state performance as item costs are slowly updated to their correct values.

### 3.5.3 Accuracy of Random Sampling

Our eviction strategy’s sampling impacts its miss-rate. Prior work [30] has studied the impacts of random sampling, and shown that the expected rank of an evicted item is \( n/(S + 1) \) where \( n \) is the number of items in the cache and \( S \) is the sample size. Increasing the sample size poses a significant burden—each sampled item’s priority needs to be evaluated, which could become expensive to compute—leading to an
Figure 3.13: Simulated performance of HC for different sampling sizes compared to finding the true minimum. The request workloads are Zipfian distributions with different skew parameters.

increase in tail latencies. However, in practice we found that this loss of accuracy is not problematic. Specifically, we measured and compared the miss rate curves for varying sample sizes on two different popularity skews (Fig. 3.13). While the smoothness of the priority distribution impacts this accuracy—and extensions like cost and expiration may introduce jaggedness into priorities—the dominating factor is how heavy the tail is and the likelihood of sampling an item from it. Sampling performs worse on the lighter-tailed distribution because there are fewer tail items in the cache, making it less likely for sampling to find them. However, for a sample size of 64, the setting that we use, the performance gap relative to full accuracy is slight.

Psounis and Prabhakar [74] proposed an optimization to random sampling that retains some number of samples between evictions. This can boost the accuracy of
random sampling, however in our tests we found the miss rate benefits to be minimal. On the light-tail distribution (Fig. 3.14), we compare performance to the suggested settings of their technique. While performance does improve for smaller cache sizes (lower hit rates), the benefits are much more limited for higher hit rates. We believe this is because tail items in a large cache tend to be young items that are less likely to be retained from prior evictions, though a more in-depth analysis is needed to confirm this. As the benefits are limited (and parameters are sensitive to cache size and workload), we did not use this optimization.

### 3.6 Related Work for Application Caching

Our introduction and subsequent discussions survey the landscape of caching work, including recency-based approaches (e.g., [28, 69, 95]), frequency-based or hybrid approaches (e.g., [57, 60]), marking algorithms and partial orderings (e.g., [28, 41]), and function-based approaches (e.g., [2, 81, 93]). All of these approaches rely on data structures and thus cannot achieve the flexibility and extensibility of hyperbolic caching.
Consider the approaches that improve recency caching. LRU-K [69] stores items in \( k \) queues and evicts based on the \( k \)-th most recent access; 2Q [50] is an example. ARC [60] automatically tunes the queue sizes of an LRU-2-like configuration. CLOCK [28] and other marking algorithms implement coarse versions of LRU that maintain a partial order. All of these strategies incorporate frequency information to balance the downsides of LRU. However, they are difficult to adapt to handle costs or other factors, due to their use of time-of-access metrics and priority orderings that are implicit in a data structure.

GreedyDual [95] exemplifies this difficulty because it attempts to incorporate cost into LRU, requiring a redesign. Cao and Irani [20] implemented GreedyDual using priority queues for size-aware caching in web proxies, and GDWheel [58] implemented GreedyDual in Memcached using a more efficient wheel data structure. The RIPQ system uses size awareness in a flash-based caching system [88]. Other cost-aware strategies have incorporated properties such as freshness (e.g., [81]), which is similar to expiration times but not as strict. In contrast to these approaches, a priority function based on frequency can easily adopt cost, expiration, or other factors.

Hyperbolic caching learns from the above and adopts a function-based approach based on frequency. Yang and Zhang [93] use a priority function that is similar to ours, but build their solution on GreedyDual by setting an item’s cost equal to its frequency. In our tests, we found that the interaction between GreedyDual’s priority queue and this frequency led to very poor performance (3-4x the miss rates of LRU). Moreover, relying on a data structure forces any function-based strategy to “freeze” an item’s priority once it enters the structure; in contrast, our priorities evolve continuously and freely. A key enabler of the latter is random sampling for eviction, which was inspired by Redis [77]. Psounis and Prabhakar [74] analyzed this technique but did not suggest any new priority functions.
Recent work in the systems community has looked at other aspects of caching that we do not address, such as optimizing memory overheads (e.g., MemC3 [37]) and multi-tenant caching (e.g., [23,75]). Hyperbolic caching does not require memory for ordering data structures, but requires space to store the metadata used to compute each item’s priority. We have not studied memory allocation across caches, but note that our framework obviates the need for separately tuned caches in some cases, for example by using our cost class extension (Section 3.3.2). This could be used, e.g., to manage pools of caches as in Facebook [68] or slabs of memory as in Cliffhanger [23].

3.7 Hyperbolic Caching Summary

In this chapter, I presented the design and implementation of hyperbolic caching. This work combines theoretical insights with a practical framework that enables innovative, flexible caching. Notably, the priority function we use reorders items continuously along hyperbolic curves. We implemented our framework in Redis and Django and applied it to a variety of real applications and systems. By using different extensions, we are able to match or exceed the performance of one-off caching solutions.
Chapter 4

Conclusion

In this thesis, I demonstrated how the typical structure, implementation, and usage of web applications can be exploited to provide better application security and better application caching performance. I detail some of the problems with application security as it exists today, and show how the Passe system can help to mitigate many of those problems by automatically applying the principles of isolation and least privilege to server-side application code and how that can be extended to application logic in the browser. In the case of application caching, I show how today’s typical caching strategies fail to exploit the regularity of modern web application behavior and demonstrate that when cache eviction strategies can be tailored to web application workloads, application throughput increases without requiring modification of application code or an increase in server resources.
Bibliography


[79] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Symposium on Security and Privacy, 2010.


