Facebook Immune System

Tao Stein
Facebook
stein@fb.com

Erdong Chen
Facebook
rogerc@fb.com

Karan Mangla
Facebook
kmangla@fb.com

Abstract

Popular Internet sites are under attack all the time from phishers, fraudsters, and spammers. They aim to steal user information and expose users to unwanted spam. The attackers have vast resources at their disposal. They are well-funded, with full-time skilled labor, control over compromised and infected accounts, and access to global botnets. Protecting our users is a challenging adversarial learning problem with extreme scale and load requirements. Over the past several years we have built and deployed a coherent, scalable, and extensible realtime system to protect our users and the social graph. This Immune System performs realtime checks and classifications on every read and write action. As of March 2011, this is 25B checks per day, reaching 650K per second at peak. The system also generates signals for use as feedback in classifiers and other components. We believe this system has contributed to making Facebook the safest place on the Internet for people and their information. This paper outlines the design of the Facebook Immune System, the challenges we have faced and overcome, and the challenges we continue to face.

Keywords Machine Learning, Adversarial Learning, Security, Social Network Security

1. Introduction

The Facebook social graph comprises hundreds of millions of users and their relationships with each other and with objects such as events, pages, places, and apps. The graph is an attractive target for attackers. Attackers target it to gain access to information or to influence actions. They can attack the graph in two ways: either by compromising existing graph nodes or by injecting new fake nodes and relationships. Protecting the graph is a challenging problem with both algorithmic and systems components.

Algorithmically, protecting the graph is an adversarial learning problem. Adversarial learning differs from more traditional learning in one important way: the attacker creating the pattern does not want the pattern to be learned. For many learning problems the pattern creator wants better learning and the interests of the learner and the pattern creator are aligned and the pattern creator may even be oblivious to the efforts of the learner. For example, the receiver of ranked search results wants better search ranking and may be oblivious to the efforts being done to improve ranking. The pattern creator will not actively work to subvert the learning and may even voluntarily give hints to aid learning. In adversarial learning, the attacker works to hide patterns and subvert detection. To be effective, the system must respond fast and target the features that are most expensive for the attacker to change, being careful also not to overfit on the superficial features that are easy for the attacker to change.

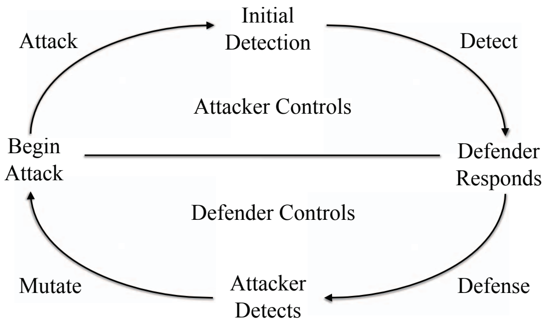

Figure 1. The adversarial cycle.
This diagram shows the adversarial cycle. The attacker controls the upper phases and the defender controls the bottom phases. In both Attack and Detect phases the attacker is only limited by its own resources and global rate-limits. During Attack, the attack has not yet been detected and is largely unfettered. During Detect, the attack has been detected but the system is forming a coherent response. This includes the time to train a model or expand the set of bad attack vectors and upload the patterns to online classifier services. The response can form continuously with some models being deployed earlier than others. During Defense, the attack has been rendered ineffective. The attacker may eventually detect this and begin Mutate to work around the defense mechanism. This cycle can repeat indefinitely. The defender seeks to shorten Attack and Detect while lengthening Defense and Mutate. The attacker seeks the opposite, to shorten the bottom phases while lengthening Attack and Detect. This cycle illustrates why detection and response latencies are so important for effective defense.

Adversarial learning is a cyclical process shown in Figure 1. An example will make the process more concrete. Several years ago phishers would attack the graph using spammy messages with predictable subject lines. The messages included links to phishing sites. They sent out these messages repeatedly from compromised accounts to hundreds of friends of the compromised accounts. The predictable text patterns and volume made these straightforward to detect and filter. To overcome this filtering, attackers obfuscated by inserting punctuation, HTML tags, and images into their messages. As well, the attackers varied their distribution channels to evade detection. The system responded to this by using mark as spam feed-

EuroSys Social Network Systems (SNS) 2011 April 10, 2011, Salzburg

back features, IP address features, and also the presence of other unusual obfuscation signatures. The IP address features were an effective response, forcing the attackers to employ botnets and global proxy networks to attack. This particular attack was destroyed, but similar attacks happen all the time. The adversarial cycle is an arms race. As detection and protection improve, the attackers in turn improve their methods.

The phase lengths in the adversarial learning cycle can vary and the goal of an effective defense is to lengthen phases the defender controls while shortening phases the attacker controls. Attacks are destroyed by making them unprofitable. In terms of Figure 1, this means lengthening the bottom phases Defense and Mutate while shortening the upper phases Attack and Detect. Together, this raises the attacker's costs to participate in the cycle and lowers their returns.

Improving the cycle requires work across all phases using multiple techniques. The Attack phase is shortened by improving detection methods: better user feedback, and more effective unsupervised learning and anomaly detection. The Detect phase is shortened by improving methods for quickly building and deploying new features and models. The defense phases Defense and Mutate are lengthened by making it harder for the attacker to detect and adapt their exploit to the defensive response. The Defense phase is lengthened by obscuring responses and subverting attack canaries. For example, a suspected phishing account can see their phishing messages, but others including the target victim cannot. The Mutate phase is lengthened by emphasizing features that are more expensive for the attacker to change. For example, using IP-related features instead of text patterns if the former are more expensive for the attacker to adapt. Shortening the length of attacker control is especially critical to defending the graph because graph-based distribution is viral and can grow exponentially. The Immune System is designed to shorten the phases controlled by attackers and lengthen the phases under defensive control.

The Immune System has two advantages over the attacker; user feedback and global knowledge. User feedback is both explicit and implicit. Explicit feedback includes mark as spam or reporting a user. Implicit feedback includes deleting a post or rejecting a friend request. Both implicit and explicit feedback are valuable and central to defense. In addition to user feedback, the system has knowledge of aggregate patterns and what is normal and unusual. This facilitates anomaly detection, clustering, and feature aggregation. The system uses these two advantages in both detection and response.

Some of the more traditional machine learning metrics do not really apply to adversarial learning in our context, or at least are less important. For example, classifier accuracy. The graph is being defended across multiple simultaneous attacks using finite resources. The goal is to protect the graph against all attacks rather than to maximize the accuracy of any one specific classifier. The opportunity cost of refining a model for one attack may be increasing the detection and response on other attacks. For these reasons, response and detection latencies can be more important than precision and recall. Even considering an attack in isolation, spending more time improving a classifier can be problematic for two reasons. Damage accumulates quickly. More accounts get compromised and more users get exposed to spam. A 2% false-positive rate today on an attack affecting 1,000 users is better than a 1% false-positive rate tomorrow on the same attack affecting 100,000 users. As well, as time progresses attacks mutate and training data becomes less relevant. Done is often better than perfect.

Protecting the graph differs from email anti-abuse in several ways. Users tend to trust Facebook identities more than email. The Facebook user interface has many different channels for communication, and new ones emerge as the interface evolves. Communication can move seamlessly between these different channels. As well, communication on Facebook tends strongly toward the real-time. These differences all have implications for the design of the Immune System and will be discussed more in Section 3.

Attacks can mutate quickly, in some cases the Defense and Mutate phases may be short, and the system must be ready to detect and respond. Due to the viral distribution of the graph, substantial damage can happen quickly when the attacker is in control. The need for a fast response has motivated much of the design described in this paper. A basic design principle is that all updates are online, classifier services and feature data providers adapt to new attacks without going offline or restarting. Responding to new attacks is a part of normal operation and normal operation should never require a service restart.

In addition to responding quickly, it is important to target features that are difficult for the attacker to detect (Defense) and change (Mutate). This differs from traditional machine-learning where the features are chosen solely on how strongly they improve the accuracy of the classifier. In general, some features of an attack are much easier and cheaper for an attacker to change than others. For example, text patterns versus IPs.

The main components of the Immune System will be described in detail in Section 4. To summarize, these are:

- **Classifier services:** Classifier services are networked interfaces to an abstract classifier interface. That abstraction is implemented by a number of different machine-learning algorithms, using standard object-oriented methods. Implemented algorithms include random forests, SVMs, logistic regression, and a version of boosting, among other algorithms. Classifier services are always online and are designed never to be restarted.
- **Feature Extraction Language (FXL):** FXL is the dynamically-executed language for expressing features and rules. It is a Turing-complete, statically-typed functional language. Feature expressions are checked then loaded into classifier services and feature tailers[1] online, without service restart.
- **Dynamic model loading:** Models are built on features and those features are either basic or derived via an FXL expression. Like features, models are loaded online into classifier services, without service or tailer restart. As well, many of classifier implementations support online training.
- **Policy Engine:** Policies organize classification and features to express business logic, policy, and also holdouts for evaluating classifier performance. Policies are boolean-valued FXL expressions that trigger responses. Policies execute on top of machine-learned classification and feature data providers. Responses are system actions. There are numerous responses. Some examples are blocking an action, requiring an authentication challenge, and disabling an account.
- **Feature Loops (Floops):** Classification generates all kinds of information and associations during feature extraction. The floops take this data, aggregate it, and make it available to the classifiers as features. The floops also incorporate user feedback, data from crawlers[2], and query data from the data warehouse.

The scalability and latency requirements are challenging. As of early 2011, there are 25B user read and write actions per day. About 20% of these are write operations. In many cases classification needs to be synchronous with user actions. In those cases latency

[1] Tailers are stream-processing programs. They read and aggregate log file data in realtime. They are called tailers because they tail logs.

[2] Crawlers are processes that take URLs and fetch their web contents.

is kept under 50ms so that the site feels responsive and interactive. Scaling the Immune System up to cover this load while meeting latency requirements has been a challenge requiring substantial engineering ingenuity.

Online updates give advantage across all phases of the adversarial cycle. Online loading of new models, policies, and feature expressions shorten response time. Better features improve both detection methods and the quality and longevity of response. Feature selection needs to consider the attacker's costs. Attackers change behavior a lot faster than people change their buying patterns.

The next section provides an overview of existing threats to the graph. Section 3 outlines the requirements to protect the graph. Section 4 outlines the basic design of the Immune System. Section 5 outlines some of the related work done on adversarial learning and traditional spam detection.

2. Threats to the graph

The social graph contains user information and facilitates connections between people to share information. It has two basic properties valuable to attackers. It stores information and it is a powerful viral platform for the distribution of information. By compromising or controlling portions of the graph the attacker can both access user information and employ viral graph distribution for spam.

Threats to the social graph can be tracked to three root causes. These are compromised accounts, fake accounts, and creepers. The response varies for these three root causes. Compromised accounts are returned to their legitimate owner and the attacker is locked out. Fake accounts are deleted. Creepers are educated and informed about how they can better use the product to not create problems or spam for others. These three root sources can abuse the same channels. For example, unwanted friend requests come from both creepers and fake accounts. Spam is a symptom that stems from these three root problems. Thinking about the root causes is necessary for solving the problems on individual channels and constructing responses.

2.1 Compromised accounts

Compromised accounts are accounts where the legitimate owner has lost complete or partial control of their credentials to an attacker. The attacker can be a phisher either automated or human, or a malware agent of some form. Spear phishing does happen where attackers may for example target the account of a poker player with many poker chips. Once the user is compromised the attacker can transfer the chips by losing in a chat room. Blame is difficult to establish here and this kind of problem generally leads to inflation in the chip money supply. Our earlier phishing classifiers made heavy use of features on IP and successive geodistance. Attackers have responded by using proxies and botnets to log in to their compromised inventory. Malware is a tough problem because the attacker is operating from the same machine as the legitimate user, so IP does not provide signal. To combat malware, the most effective mechanism we have discovered is to target the propagation vector using user feedback. Attackers can also try to game user feedback features. That is combatted with reporter reputation and rate limits.

Attackers build social credibility and graph structure around objects by attaching fake accounts to them. Fake accounts on average have lower account age than legitimate accounts because the system detects and disables them. If the attackers cannot use fake accounts to pull their spammy objects into the graph then they typically draw from their inventories of compromised accounts. Compromised accounts are typically more valuable because they carry established trust. The attackers will deploy them more carefully.

Phishing Phishing attacks aim to steal the login credentials of users. They typically spread by hijacking the user trust embedded in the graph. Successful phishing attacks are particularly harmful because attackers gain control over trusted nodes in the graph. They then exploit these trusted connections for financial gain and to further their own purposes.

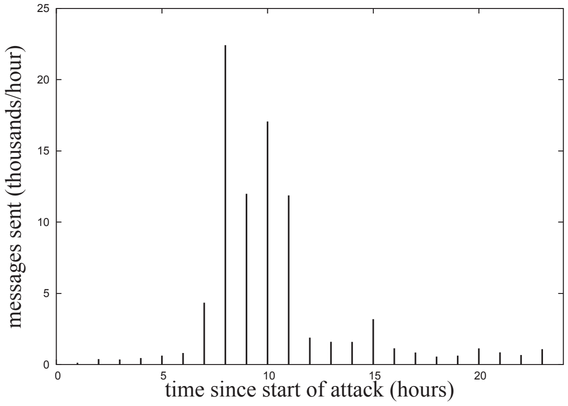

Figure 2. Timeline of a phishing attack on Facebook.
This illustrates the different phases of an attack cycle and how an attack can increase in volume rapidly. The graph plots the number of phishing messages per hour for a coordinated phishing attack. The attack begins at time 0, with the attacker starting with a small set of accounts, sending out phishing messages. The messages contain links to a phishing page. Some users go to the page and reveal their credentials to the attacker who then gains control of their account and then uses it in the campaign. Initially the attack is small, but gradually gains accounts. The system detects the attack in hour 8 and forms a response, applying it immediately and improving the response to return the accounts to their owners, driving down the attack volume as the attackers lose their compromised account inventory.

Users are more vulnerable to phishing on social networks than they are on email. Social network identities have social context, attached photos and other metadata, and also authentication. Attackers use these in social engineering attacks to improve the pull of their lures. People have been conditioned to be wary of links in email, but tend to put more trust in social network messages from their friends and the links within. This trust is a target for manipulation by phishers.

Malware Malware is a software agent that resides on a user's computing device and uses that device to piggyback on the user's credentials. Once installed on a device, the malware can hijack the user session and send messages on behalf of the user. These messages may contain the payload for the virus to replicate itself or more often a link to a download site that distributes the payload. Users are often unaware their device is infected with malware. The agents are often silent and cloak themselves to avoid detection and cleaning.

Since the attacker controls the physical machine of the user, it is difficult to both detect and prevent malware. A common response to automated attacks is to require Turing tests. For example, CAPTCHAs [von Ahn 2003]. However, some malware variants have found ways around this. For example, Koobface malware variants have been observed to take the challenge and forward it to the user on the same machine, enclosed in a window box warning that the operating system will shut down if the user does not solve the challenge. Many users comply and Koobface forwards the solution back to Facebook, solving the challenge. Using this method

Koobface has been observed to achieve solve rates comparable to humans.

As mentioned above, the most effective mechanism against malware has generally been user feedback on the channel the malware is using to spread. For example, users marking posts as spam. This is used in combination with a basic message classifier. Users are identified as infected when they send many messages marked sketchy by the classifier and many of their friends mark these as spam. The user marking as spam is a strong signal in combination with sparse text features and basic features on embedded URLs.

2.2 Fake accounts

Facebook is for real people and the Facebook ID is not intended to be a pseudonym or handle. Each person has at most one account. The authenticity of user IDs is one of the core premises of Facebook. Significant numbers of fake accounts undermine this authenticity and corrode user trust in the network. Fake accounts can be created non-maliciously by people that just want an extra account. Much more commonly, fake accounts are created by attackers that want to compromise and influence regions of the graph.

Fake accounts are created by both scripts and raw labor as inventory to attack the graph. Fake accounts are created mainly for three reasons. Attackers create parallel fake accounts to overcome rate limits associated with individual accounts. They also use parallel accounts to friend or like objects en masse and thereby boost the reputation or ranking of those objects. Finally, they use fake accounts as fake identities to phish and spam real users.

Fake accounts have limited virality because they are not central nodes in the graph and lack trusted connections. They also have no unique data or history. For these reasons, they are much less valuable to attackers than compromised accounts. Attackers will protect their compromised account inventory much more carefully than their fake account inventory.

Note that the value of a fake account is measured over its lifetime. Catching a fake account early in its lifetime thereby decreases its value. By catching fake accounts early before they can be productive, the Immune System drives down the total resources the attacker can rationally justify investing. Ideally, this is a self-reinforcing loop that kills an attack.

2.3 Creepers

Creepers are real users that are using the product in ways that create problems for other users. One example of this is sending friend requests to many strangers. This is not the intended use of the product and these unwanted friend requests are a form of spam for the receivers. Another example is posting spammy chain letters in statuses to get broad feed distribution.

Chain letters typically use social engineering to motivate receivers to repost. The letters claim that if the receiver does not repost something bad will happen, like they will lose their account or if the receiver does repost something good will happen, like money will be donated to charity. Chain letter volume can explode when spread using the powerful viral channels of Facebook. In the past, they have been observed to reach 1-5% of total user communications in minutes.

Chain letters can cause two problems beyond simply bothering the receiver. They can motivate users to take damaging actions on false pretenses and they can create a global misinformation wall that hides critical or time-sensitive information.

In most cases it seems impossible to trace a chain letter back to one originator. The memes probably exist elsewhere and enter the graph through multiple entry points, mutate, leave, re-enter, and so on. Chain letters exploit social engineering to trick otherwise well-behaved Facebook users into propagating the attack. As with other creeper attacks, the best long-term answer is education. In the short-term other mechanisms can be used against chain letters specifically. For example, fuzzy n-gram matching or other forms of locality-sensitive hashing on text. These techniques tend to work well primarily because chain letters are not adversarial in the strong sense of say, phishing.

2.4 Spam

Spam is a symptom of these three root problems. The site provides numerous channels for interaction between users. All of these are targeted by attackers to distribute spam. They generally flow to the channel with the least protection and highest reach. Some spam is obvious and universally spammy. For example, phishing links. Other spam is more difficult to call. Most of the stuff in this second category tends to come from creepers. Spam from phished and fake accounts is straightforward to deal with – block or move to a spam folder. Creepers on the other hand are often just unaware or acting on bad information. They can benefit from education and better information.

In many cases the offender is unaware that their actions are considered spammy by others. For extreme and persistent cases, Facebook educates users on how to use the different features of the site so they send out less unwanted information. For combatting unwanted friend requests, education has had a substantial positive impact on user behavior. By definition, spammy friend requests happen between two users that are not friends. When two users are friends and the behavior of one is bothering another, ideally the two can resolve conflict without system involvement. If a user reports a friend as abusive, under some conditions the system will encourage the two users to communicate directly to resolve the conflict. User education can help reform creepers bothering people that are not their friends, but ideally conflicts between friends can be solved directly between the two friends.

Defining spam is difficult on a site with a global user base. Different cultures have different social norms around communication. Acceptable behavior in one region may be interpreted as unwanted contact in another. This makes a uniform definition of spam difficult. In general, the working definition of spam is simply interactions or information that the receiver did not explicitly request and does not wish to receive. Both classifiers and the educational responses need to be tuned for locale and user. Cultural breadth necessitates different models and responses with regional and cultural features.

3. Design principles for protecting the graph

To protect the graph the Immune System runs classifiers to block and respond and anomaly detection to detect new and mutated attacks. Developing, deploying, and operating these classifiers has a number of challenges. Attacks mutate across different channels within a large user-interface surface area. The system must defend against these attacks while meeting severe scalability and latency requirements. This section discusses several of the important system requirements.

3.1 Detect and respond quickly

The virality of Facebook communication can be used by attacks to grow quickly. Figure 2 shows the spread of a phishing attack on Facebook. The attack starts as a low volume attack. Within a couple of hours, the attack compromises new accounts and grows exponentially. Once the system detects and blocks the attack, the number of messages declines. To minimize damage the system must detect new attacks and respond quickly.

3.2 Cover a broad and evolving interface

Another complicating issue is the breadth and complexity of the user interface. On Facebook users do not only send messages to each other. They send chats, tag photos, write comments and interact on numerous other channels. New channels develop all the time as the user interface changes. Not only are there many different channels for communication and interaction, but users move fluidly between them. The breadth and fluidity of communication necessitates sharing of feature signals across channels. As well, attacks can change and develop in unexpected ways as the user interface changes. Changes in the interface can sometimes break models by changing the meaning of a feature. The system must monitor model quality and initiate retraining when necessary.

3.3 Share signals across channels

Facebook has many different channels for communication. For example, chat, messages, wall posts, public discussion and friend requests. Users can communicate privately, publicly, or anywhere in between. Communication can cross between these different forms and a logical conversation can cross several different channels. Like users, attacks use many different channels. For the system to be effective it must share feedback and feature data across channels and classifiers.

3.4 Classify in realtime

The realtime nature of communication and interaction on Facebook motivates realtime classification. On Facebook, users communicate in a pattern more similar to online chat than email. This is facilitated and encouraged by the user interface design. A user of an email client or web portal may check in on their email periodically, perhaps ranging from a couple times a day to a few times an hour. In this access pattern there are gap latencies between writes and reads that give breathing room on classification latency. In contrast to email, the Facebook user interface integrates email and chat and also can be configured by the user to push event notifications to external addresses and devices. For example, users can configure their account to notify them via SMS on various events. These notifications are popular and decrease the latency on interaction. Interaction on the site biases towards realtime, meaning classification must be realtime to be effective.

Some classification responses can be delayed, batched, and take longer. However, many classifications do need to be synchronous with user actions. There are several reasons for this: the communication itself is synchronous and realtime (like chat), the response is more meaningful and coherent with immediate action context (blocking an unwanted friend request), and downstream load would damage the system without filtering (some actions consume substantial resources if they pass). The system must meet these realtime latency constraints.

4. Design

As discussed above in Section 3, a system to protect the graph has a challenging set of requirements. It must react quickly to new threats, have realtime access to rich and timely feature data, scale with low latency, and provide reliable service.

Figure 3 shows the basic architecture of the Facebook Immune System and describes the flow of a typical user action. The policy layer evaluates the policies, triggering classification and feature extraction, then maps to a response. After response, the system updates historical feature data using the three feature loops.

Policies express classification as a feature function named *ClassifyScore*. It takes the names of the classifier and corpus as parameters and routes the feature map to the corresponding algorithm and model. This function evaluates to a certainty value and that value is used within the policy expression in the same way as any other feature value. At the level of policies, classification looks like other features. Policies use classifiers as features, classifiers use features in their models, and classifiers can use features that are not classifiers. Figure 3 illustrates these component relationships with classification between policy and feature extraction and also a direct arrow from policy to extraction.

After classification and policy evaluation complete, the accumulated feature map is passed to a counting step that updates feature loop counters. Finally, the feature map flows into a log where it is processed by a set of tailers. These tailers process the stream of feature maps coming out of the classifiers. They do a number of different things; anomaly detection, clustering, further feature extraction, and aggregation and summation of features. Finally, the feature map flows into the data warehouse, to facilitate larger, parallel feature synthesis and joins with other tables.

The faster the system responds the less time the attacker has to compromise accounts, create fake objects, and send spam. New models are loaded online, training is online, policies are updated online, and new features are created and updated online. Some of these operations happen often. For example, some tailers analyze the feature logs for new patterns and anomalies. They feed data into online models to train them. Depending on the rate of anomalies and attacks, the tailers may be adding new data every second or more frequently. Restarting the services would be infeasible. As discussed in Section 1, new attacks are a part of normal operation and normal operation should never necessitate a service restart. Fast online response drives down the attacker's yield.

This section elaborates on the design details of the individual Immune System components.

4.1 Classifiers

The classifier modules are compiled into a Thrift[3] service. They can also be compiled directly in with the client program or run as a separate service either locally or remotely. One advantage of running the services remotely is that many clients can share a smaller number of services, amortizing their memory, CPU and other resources.

Classifiers have internal state; feature expressions, models, and policies. Classification state in the form of counters and feature values is not stored in the services but rather is shared across all services in a distributed shared memory layer described more in Section 4.4.1.

4.2 FXL

Classifiers and policies compute decisions on feature values. Features are often derived, meaning that they are expressed in terms of operations on and combinations of other features. Initially, all features were flat names with static definitions inside the service code. Responding to attacks, new classifiers or policies were regularly required with new features. This necessitated not only a model update, but also implementing new features in the service code, reviewing that code, testing it, and pushing the new code to thousands of classifier machines. This imposed substantial overhead, introduced opportunities for error, created redundant code, and slowed attack response time.

As well, experimenting with features was not easy. The inconvenience of experimentation meant that features were often not as good as they could be. The quality of features affects the lengths of all phases of the adversarial cycle. Making feature experimentation easy encourages exploration and thereby leads to improvements. Part of building an effective defense structure is making it easy to experiment and try out ideas quickly, while protecting the system from the consequences of experimental errors. A low cost barrier to discerning the good from bad ideas is central to progress. Deploy-

[3] Thrift is a network RPC protocol. See `thrift.apache.org`.

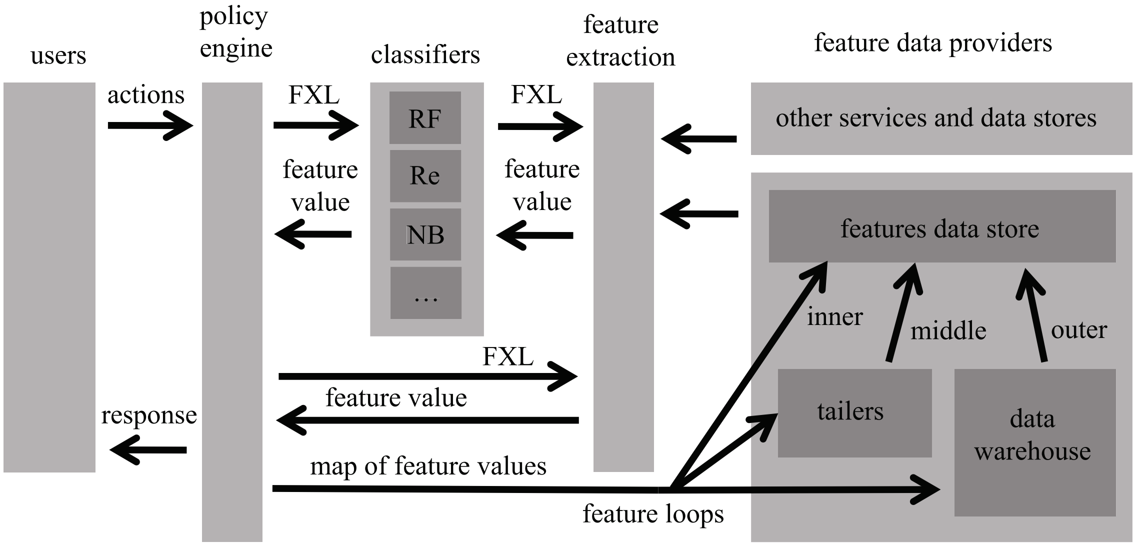

Figure 3. High-level design of the Immune System.
This diagram outlines how a user action flows through the system. Various classification algorithms are shown with the labels RF (Random Forests), NB (Naive Bayes), and Re (Regression). Time starts at the top. A user action flows in with some basic feature values. The policy layer runs all relevant policies on the action. Policy evaluation triggers data fetching. Requests are batched for efficiency. Caching is done with memoization at the language evaluation layer. There is little locality between actions. Once all the features have been collected, the system evaluates the policy and maps it to a response. This completes the user action. Following that, realtime counters are updated and the feature map is written to a log. This log flows into tailers and the data warehouse. These feature loops provide classification history and sharing. The longer feature loops have access to progressively more data, but at increasing latency. Inner (counters) has latency of 10-15ms, middle (tailers) has latency of 5-10s, and outer (warehouse) has latency of 1 day.

ing features monolithically within the service code made it difficult to experiment with features, thereby hindering defense of the graph.

FXL was designed to improve response time and make it easier to develop features. The key observation motivating FXL is that many features are in fact derived from a smaller set of basic features. This means that a language can be used to form new features by combining and operating on basic features. FXL allows us to create and deploy features quickly, makes features engineering easier, and eliminates redundant code.

FXL is strongly typed to facilitate static checking and functional to facilitate terse expressions and code reuse. All expressions evaluate to typed values and the types are known statically independent of the runtime values. It has exceptions and a try-catch mechanism for error handling. Higher-order functions and dynamic function abstraction (lambdas) facilitate code re-use.

FXL preserves the correctness benefits of compiled code while allowing features to be loaded online. All expressions are parsed and type-checked and only loaded online if they pass those checks. This catches errors early on before they reach production.

A functional language has several advantages for our use. Expressions can be easily represented as trees. Trees lead to straightforward manipulations for execution and parallelization. As well, the lack of side-effects and global variables enables memoization of subtrees, eliminating duplicate I/O and computation.

Building a new language from scratch can be risky so we patterned FXL after a well-defined and mature language. We chose Standard ML because it is strongly-typed and functional and has many of the features like higher-order functions and lambdas that are helpful for deriving features quickly and correctly.

Here are some examples of FXL expressions. Here is the max domain spam scores of all domains in a message text:

```
Max(Map(DomainSpamScore, ExtractDomains(Text)))
```

The number of common liked pages between a sender and receiver is represented as:

```
Count(Intersect(LikedPages(Sender),
                LikedPages(Receiver)))
```

To make FXL easy to use from anywhere, the interpreter is accessible directly through the networked classifier services. This means that clients can make calls to the services to execute these feature expressions and get values back. If a client is concerned about the Thrift RPC round-trip cost, it can run a service locally or bypass the service entirely and statically compile with the FXL interpreter. A command-line interpreter and web interface also exist to facilitate feature experimentation.

4.3 Policy layer

The policy layer gives analysts control over how classifiers are applied and the responses they trigger. Policies are boolean-valued rules expressed in FXL to leverage all its data access and manipulation primitives.

The decision about how and when to respond can depend on business or policy considerations. For example, an action in one region might be more creepy or undesirable than in another region. Another example would be applying a more aggressive spam classifier to pages depending on their admin preferences. Business logic or policies of this form do not belong in learned models and would only damage their performance.

The policy layer also provides a mechanism to evaluate classifier performance. The holdout mechanism divides the affected users into two groups, a regular group and a holdout group. The holdout group can be much smaller. The responses generated from the rule are applied to the regular group but suppressed for the holdout group. Follow-up evaluation can show the differences between the two groups and how the classifier was performing. For example, to what extent where friend requests labeled unwanted by the classifier actually rejected by the receiver?

Finally, policies give fine-grain control over the application of the classification hypersphere. This is important because of the many communication channels. Classification may make sense on some slice of the channels, but not all. Adding rules to gate classification on a feature can significantly boost the precision with little sacrifice on recall. In some cases noisy training data may generate a classifier with poor accuracy on the general testing data, but excellent accuracy when narrowed down by a policy.

The following example shows how a policy can be used both to combine together different classifiers and also to control the application of those classifiers to one particular channel.

```
And(IsChannel("messages"),
 And(GreaterThan(Count(ExtractURLs(Text)), 0),
  And(
   GreaterThan(
    ClassifyScore("fakers", "2011-03-15"), 0.41),
   GreaterThan(
    ClassifyScore("bad_urls", "2011-03-14"), 0.74)
 )))
=> SpamFolder
```

This policy will be evaluated on every message send. If it evaluates to true, the message is moved to the receiver's spam folder. Otherwise, the message passes through to their inbox. If desired, sampling can be expressed within a policy.

4.4 Feature Data Providers (Providers)

Classifiers benefit from high-quality data, lots of data, and recent data. Feature selection can improve classifier accuracy more than picking the best algorithm. Accuracy benefits from clean, relevant, and timely feature data.

Getting feature values in realtime can be challenging. Data may be located in a data warehouse, in an online database, behind a service, or in some form of distributed shared memory cache. There may be cross-language issues with the data service written in one of many languages. The data may even be distributed and unsynthesized and require aggregation before use. The feature data provider interface abstracts these different data sources. It gives the FXL runtime a standard interface through which to efficiently fetch data. Cross-language issues are solved by the provider implementation either using a foreign-function interface or communicating with the data provider over Thrift, an RPC protocol.

There are a number of implemented feature providers. The entire Facebook PHP codebase is compiled and statically linked behind this interface using the HipHop[4] compiler. This gives FXL access to any data represented in the site's PHP frontend code. Other providers in production include interfaces to search indices, various Memcache[5] data layers, Thrift services, and data generated by large Hive[6] queries running on the data warehouse. As well, there are providers for realtime counters on IPs and other features. Together these providers provide FXL with the basic feature values. FXL expressions build derived features on these basic values using functional folds, maps, and various other operators, as discussed above.

The provider interface makes it easy to integrate new providers of feature data, but not all providers are the same. Latency and reliability can vary. Section 4.5 describes how the Immune System deals with performance and reliability heterogeneity across data providers.

4.4.1 Feature Loops (Floops) provide history

To effectively combat mutating attacks, the system must be able to track and store new features dynamically. Every action may trigger multiple classifications. For each classification the Immune System extracts a wide variety of data about the action. Feature loops aggregate feature data across actions and store it as features for classification. Conceptually, they provide classifiers with a shared memory about past observations and classifications.

The Immune System implements three related mechanisms to aggregate features for actions. To keep the Attack phase short, aggregate features should be available for classification quickly. Aggregation across larger data sets and application of complex operators extend the Mutate phase by making it harder for attackers to adapt to these features. As the data set and operator complexity increases and spans more data, feature aggregation latency rises. The three feature loops vary across the spectrum of feature complexity and latency.

Inner The inner loop contains low-latency increment and decrement counters. It provides a mechanism to count on values of any subset of features. A subset of feature names is specified for counting. The counters store the number of occurrences of each possible combination of values for these features for a defined period of time. For example, the number of times a URL has been posted on a channel in the past hour. The rate of posting can be used as a feature in classification. On every classification call the appropriate counters are read from a Memcache tier shared across services, updated and written back to Memcache. For classification this data can be extracted using FXL with the Memcache tier as the data provider.

Latency between an classification and the appropriate counter values being updated is within 10-15 ms. The updates themselves are not computationally expensive allowing a large number of counters to be maintained. Currently the Immune System stores counts on 500 separate feature sets.

Middle The middle loop provides a mechanism to apply more powerful and complex operations on the feature logs beyond maintaining counts. This is implemented as an FXL extension which can talk to Memcache allowing easy addition of new information that can be later extracted for classification. Memcache is used to store a set of features about IPs and URLs. On every classification FXL expressions are executed which generate new values for these features. The new values are written back to Memcache for use in classification. To avoid race conditions all updates related to a specific IP or URL are sent to the same machine for processing. Routing data to the appropriate machine and evaluating the FXL expressions introduces latencies of 5-10 s.

Outer Some features require applying aggregate functions across many actions. For example, a useful feature for detecting suspicious URLs is the unique number of users who have posted the URL. To compute this feature all posts from the previous day need to be aggregated. The Immune System uses Hive to process data for all actions daily and extract useful features. These features are uploaded into Memcache to be used in realtime classification. Feature latencies are typically up to a day for these features.

4.5 Reliability and performance

The provider layer has been one of the major challenges for Immune System reliability. FXL execution depends on the provider

[4] `https://github.com/facebook/hiphop-php/wiki/`

[5] Memcache is a distributed memory object caching system. See `memcached.org`.

[6] Hive is a data warehousing infrastructure and query language built on top of Hadoop. See `hive.apache.org`.

systems. These systems can fail or return invalid feature data. Failure and periodic anomalies are something of a fact of life in a complex distributed system, and even more so in one scaling so quickly.

The system has several mechanisms for detecting and protecting the system from these faults. Unit tests run on code changes before they are committed to trunk. All the unit tests run every night after the nightly build of the entire world. Dynamic checks run every few minutes checking that providers return correct expected values. Realtime counters on every basic feature monitor whenever a provider fails to return data. Finally, feature engineers and model designers have control over failures. FXL has a try-catch exception mechanism. If a provider fails to return a requested feature value, FXL throws a FeatureNotFound exception. This exception can be caught by the expression. From the catch, the expression can use a default value, another feature, or rethrow the exception. Together, these mechanisms detect and control feature failure.

Dynamic model and feature loading complicates reliability. Adding new models to the system can change the number of features required by classification. This can impact CPU utilization and classification latency in unexpected ways. Latency for accessing a feature can vary by several orders of magnitude, from microseconds to milliseconds. Some providers store data locally, some in Memcache, some in Thrift services, and some even access databases. In the case of the HipHop PHP provider, the latency can be opaque and difficult to predict statically. For example, the PHP code might do some computation or access Memcache or even issue a database query. The system provides a staging environment called P1 to test changes before deployment to production. P1 contains an independent tier of online classifier services. A fraction of classification traffic is sent to P1 asynchronous with user actions. This fraction can be tuned to evaluate the impact of load changes. P1 measures the latency and reliability of each feature. These observations are used by model designers to test the performance of a classifier before deployment and decide whether a model or feature should be added to the main production tier.

5. Related Work

This paper has outlined the threats to the graph and the system we have built to protect it. Protecting traditional email systems has been studied for some time. Blanzieri [Blanzieri 2008] provides an overview. Some research has investigated improved machine learning methods or features. Carreras and Marquez [Carreras 2001] show how Boosting Trees can improve classification. Hao et. al. [Hao 2009] explore non-text features for improving email reputation systems.

As social media has become more popular it has been increasingly targeted by sophisticated attackers. Unlike traditional email abuse, attacks on social media use multiple channels. Heymann et al. [Heymann 2007] overview approaches to the problems of spam in social media. Malware is one of the most difficult problems. Koobface was one of the first social malware variants to emerge, spreading via Facebook, Twitter, and other sites. Thomas et al. [Thomas 2010] analyze Koobface and the general problem of social malware.

Protecting the graph is an adversarial learning problem. Dalvi et al. [Dalvi 2004] and Lowd and Meek [Lowd 2005] discuss methods for adversarial learning and classification. Ma et al. [Ma 2009] and Whittaker et al. [Whittaker 2010] address the problem of optimizing classification methods or reducing feature spaces to improve the classification latency. However, neither consider the performance and reliability issues of fetching features from numerous heterogeneous data sources. Ma et al. [Ma 2009] use lexical and host-based features of the URL and discuss lightweight classification methods.

6. Conclusion

This paper overviews the threats to the graph and describes the system currently in production protecting the Facebook graph. The main contribution of this work is an integrated system for machine learning on an adversarial problem. The system is scalable and responsive. New models and new features can be added online, and the system generates many signals that can be used as feedback in classifiers or in external systems.

Other anti-abuse and adversarial learning problems are likely to benefit from the ideas behind the Facebook Immune System, primarily the focus on fast detection and response, sharing data across channels, and the integrated feature feedback loops. Adversarial learning is challenging because attackers can detect defenses and mutate their exploits quickly. There is more work to be done improving mechanisms to lengthen defense control and shorten attacker control. An appealing future direction lies in realtime training of machine learning models that better leverage both user feedback and anomaly detection.

Acknowledgments

Many Facebook engineers have contributed to the systems described in this paper. In addition to the authors, substantial contributions have been made by Bhal Agashe, Zhimin Chen, Daniel Gibson, Pedram Keyani, Clément Genzmer, Mark McDuff, Allan Stewart, Hossein Bokharaei, Jon Coens, and other members of Facebook's Site Integrity team. Mukund Narasimhan, Carlos Bueno, Yoann Padioleau, and David Molnar gave helpful feedback on drafts of this paper.

References

[Blanzieri 2008] E. Blanzieri and A. Bryl. A survey of learning-based techniques of email spam filtering. *Artif. Intell. Rev.*, 29:63–92, March 2008.

[Carreras 2001] X. Carreras and L. Márquez. Boosting trees for anti-spam email filtering. In *Proceedings of RANLP-01, 4th International Conference on Recent Advances in Natural Language Processing*, Tzigov Chark, BG, 2001.

[Dalvi 2004] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 99–108, New York, NY, USA, 2004. ACM.

[Hao 2009] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with snare: spatio-temporal network-level automatic reputation engine. *USENIX Security Symposium*, page 101118, 2009.

[Heymann 2007] P. Heymann, G. Koutrika, and H. Garcia-Molina. Fighting spam on social web sites: A survey of approaches and future challenges. *IEEE Internet Computing*, 11:36–45, November 2007.

[Lowd 2005] D. Lowd and C. Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 641–647, New York, NY, USA, 2005. ACM.

[Ma 2009] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 681–688, New York, NY, USA, 2009. ACM.

[Thomas 2010] K. Thomas and D. M. Nicol. The Koobface botnet and the rise of social malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 63–70. IEEE, October 2010.

[von Ahn 2003] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *EuroCrypt*, 2003.

[Whittaker 2010] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *NDSS*, NDSS '10, 2010.