IVD: Automatic Learning and Enforcement of Authorization Rules in Online Social Networks

Paul Marinescu†, Chad Parry†, Marjori Pomarole†, Yuan Tian‡, Patrick Tague‡, Ioannis Papagiannis†

†Facebook. {pau, cparry, mpomarole, yiannis}@fb.com
‡Carnegie Mellon University. {yuan.tian, patrick.tague}@sv.cmu.edu

***Abstract*—Authorization bugs, when present in online social networks, are usually caused by missing or incorrect authorization checks and can allow attackers to bypass the online social network's protections. Unfortunately, there is no practical way to fully guarantee that an authorization bug will never be introduced—even with good engineering practices—as a web application and its data model become more complex. Unlike other web application vulnerabilities such as XSS and CSRF, there is no practical general solution to prevent missing or incorrect authorization checks.**

In this paper we propose INVARIANT DETECTOR (IVD), a defense-in-depth system that automatically learns authorization rules from normal data manipulation patterns and distills them into *likely invariants*. These invariants, usually learned during the testing or pre-release stages of new features, are then used to block any requests that may attempt to exploit bugs in the social network's authorization logic. IVD acts as an additional layer of defense, working behind the scenes, complementary to privacy frameworks and testing.

We have designed and implemented IVD to handle the unique challenges posed by modern online social networks. IVD is currently running at Facebook, where it infers and evaluates daily more than 200,000 invariants from a sample of roughly 500 million client requests, and checks the resulting invariants every second against millions of writes made to a graph database containing trillions of entities. Thus far IVD has detected several high impact authorization bugs and has successfully blocked attempts to exploit them before code fixes were deployed.

I. INTRODUCTION

Modern online social networks (OSNs) handle large amounts of user data. These data are often generated by users, are associated with their accounts and are subject to access control rules governing who can read, create, modify, and delete them. Because OSNs enable many types of user interactions, with different levels of permissions, writing and enforcing these rules quickly becomes nontrivial. Developers need to flawlessly consider all possible interactions and correctly implement the appropriate checks while at the same time iterate quickly to satisfy business needs.

OSNs are constantly evolving, with new features being added regularly, often in an arms race to offer their users an improved experience and more ways to express themselves, which often concretizes in new types of interactions. For example, users may use private messaging for a one-to-one or a group conversation, in which case only the participants should be allowed to send and receive messages associated with the conversation and only participants or the conversation moderator, depending on policy, should be allowed to add people to the conversation. Furthermore, messages may be edited in a short time interval after they were sent, but only by the same person who sent them. Users may also interact by posting content to their personal page, usually unrestricted, or to a group, usually only after being explicitly accepted in the group. They may also create connections (e.g. *befriend*, *follow*, *connect*, become a *fan*, *add to circles*) with other users, which often gives them additional ways to interact.

Authorization becomes even more difficult when multiple types of entities and delegation are involved. For example, an OSN can support personal users, businesses, and a many-to-many *business administrator* relation between them. Users connected by the business administrator relation are authorized to act on behalf of the business in matters such as changing the business address or answering customer messages. In addition to this, a different relation, *business owner*, allows users to merge two businesses into a single entity. Proper authorization for the merge requires checking that the logged-in user is the owner of both businesses.

Failing to perform correct authorization checks leads to authorization bugs. Attackers can exploit them to impersonate other users, perform actions on their behalf and gain access to data. While all businesses can be negatively affected by such bugs, OSNs are particularly sensitive: they contain large amounts of user data, e.g. their pictures, personal interests, job applications, physical location, and are home to their users' online personas. In consequence, an OSN's reputation depends heavily on user trust.

However, missing or incorrect authorization checks are a common issue. The Open Web Application Security Project (OWASP) lists authorization bugs as the cause of two of the top ten most common and important classes of web application vulnerabilities [1]. In addition, authorization bugs are easy to exploit if found. Typically, a malicious actor approaches such an attack by trial and error: they first understand the API of a web application by inspecting its normal functionality, they identify the arguments that these APIs receive, and finally send requests with systematically modified arguments and check whether the application performed any action on the object identified by the recently modified argument. The low success rate of each attempt is balanced out by the short time needed to devise the attack, the possibility of automating it, and by its simplicity and accessibility.

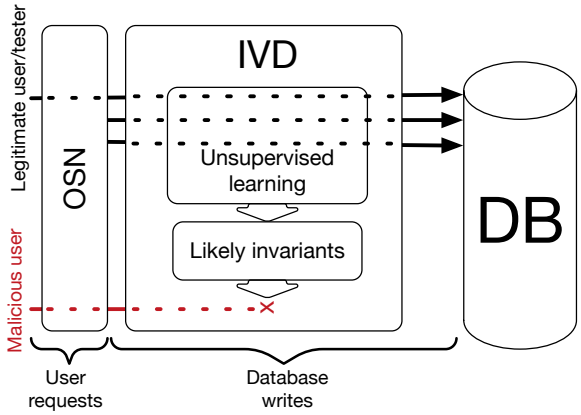


Fig. 1: INVARIANT DETECTOR distills normal behavior into invariants that it then uses to blocks malicious requests.

We believe that writing, maintaining and consistently enforcing correct authorization checks in a quickly-evolving, complex application is difficult, even with privacy frameworks and engineering best practices. Developers need a holistic understanding of the application and good programming discipline to avoid introducing bugs, and new contributors need to be particularly careful to fully understand the privacy implications of their changes.

To mitigate this problem, we propose INVARIANT DETECTOR (IVD), a *defense-in-depth* system based on dynamic invariant detection, that provides a safety net against missing or incorrect authorization checks. As shown in Figure 1, IVD intercepts requests made by an OSN to its database. It mines likely invariants from the database writes the application code performs and blocks database writes which break existing invariants. IVD has a short learning period, usually covered by internal testing, dogfooding, or a pre-release period, making it ready to act by the time a new product feature is made available to users. As we explain in more detail later (§III), IVD's design allows it to adapt automatically to OSN changes by continuously learning invariants, without requiring manual intervention or explicit training data sets.

While dynamic invariant detection has been previously proposed for testing [2], [3], [4] and attack detection [5], [6], most approaches have targeted the network or application layer. Our experience shows that inferring invariants at the database layer may instead offer more advantages by allowing both expressive invariants and unprecedented scalability.

OSNs pose unique challenges to a dynamic invariant detection system. Some challenges stem from the sheer scale of an OSN's day-to-day operations; many popular OSNs have well over 100 million active users, going up to 1.8 billion. This leads to a large number of requests that have to be checked in real time and a potentially huge invariants learning corpus. The problem is compounded by the highly interconnected nature of an OSN's data, which can result in multiple objects being involved in the final decision of an authorization check, a characteristic that has to play a role in the invariant design. The scale also reflects in the impact of false positives, causing even false positive rates as low as 0.000001% to be impractical due to the number of affected users and false alarms requiring human attention.

Other challenges have to do with the complexity of an OSN's codebase. Any practical invariant-based system working at this scale must provide, along with its alerts, sufficient information to allow engineers to quickly understand whether they are facing a bug or a false positive. In case of a bug, the information should facilitate debugging, while in the case of a false positive there must be a straightforward way to blacklist the spurious invariant. Black-box systems [7] have inherent difficulties to offer this information as they are completely agnostic to the application logic. IVD, on the other hand, is positioned between the OSN's code and its database. It can both make use of the database schema to get insights into the data, and access and report the application's state, e.g. the call stack at the time of a data access.

This paper presents our experience with running an invariant detection system at the scale of an OSN and the set of trade-offs between performance, invariant complexity and mining technique required to make this possible. To our knowledge, our system handles orders of magnitude more data than previous invariant detection systems.

The main contributions of this paper are:

- A scalable distributed dynamic invariant detection system for highly interconnected data.
- A two-step invariant generation mechanism and a set of design and implementation choices that allow the system to scale and to achieve negligible runtime overhead.
- A set of domain-specific *enforcement excuses* that tackle the inherent susceptibility to false positives of invariant detection systems.
- Results showing that dynamic invariant detection can effectively identify incorrect authorization checks and prevent attackers from exploiting them in a real-world OSN.

This paper is organized as follows: we start with background information pertaining to the data model that we target and existing practices and techniques for authorization enforcement (§II). We then provide a high level description of IVD (§III), after which we present implementation choices, focusing on scalability and performance (§IV). We continue by evaluating IVD's effectiveness and performance (§V), and finally, we present related work (§VI) and conclude (§VIII).

II. BACKGROUND

In this section we discuss the classic approach for avoiding authorization bugs in web applications and where this fails. We specifically look at a graph data model, we then describe how an attacker can discover and take advantage of authorization bugs and what IVD does to prevent this.

A. Graph Data Model

A graph data model structures data into interconnected *objects* that form a graph. This organization of data is useful

for any application domain that needs to efficiently generate fine-grained customized content from highly interconnected data, and is particularly popular among OSNs [8], [9], [10]. In this paper we use an *attributed graph model* [11], which we briefly describe below for completeness.

The two fundamental entities of a graph data model are *objects* and *associations*, corresponding respectively to the nodes and edges of the underlying graph. Objects have a unique identifier (ID) and a type (OTYPE) while associations are identified by their source object (ID1), destination object (ID2), and type (ATYPE). In addition, in attributed graph models, both objects and associations may have properties in the form of KEY $\mapsto$ VALUE pairs. To summarize,

Object: (ID) $\mapsto$ (OTYPE, (KEY $\mapsto$ VALUE)*)
Association: (ID1, ATYPE, ID2) $\mapsto$ (KEY $\mapsto$ VALUE)*

where the Kleene star (*) denotes *zero or more*. In other words, the graph data model maps object identifiers to their type and properties, and maps object pairs along with an association type to the properties of that association. When an object or association does not exist, the mappings are not defined.

An OSN might use objects to represent users, business profiles, groups, or photos, and associations to represent owner relationships, friendships, follower-followee relationships, or like relationships.

The fundamental operations supported by the graph database are retrieval, creation, deletion, and mutation of objects and associations.

While this paper focuses on a graph data model, the ideas underpinning IVD naturally transfer to other data models which use similar concepts. For example, a relational database model replaces objects with *records*, object ids with *primary keys*, object attributes with *attributes* and relationships with *foreign keys*, while an object-oriented databases exposes similar concepts through *objects*, *object ids*, *attributes* and *pointers*.

B. Reads vs. Writes

To understand IVD's applicability, it is important to distinguish between how authorization checks are performed for read and write operations. The main insight into their difference is that authorization checks often rely on information contained in the graph, and therefore fundamentally need to happen after reads. In other words legitimate object reads may happen even if the user triggering them is not authorized to see the information being read. Writes, on the other hand, should always be preceded by any needed authorization checks since writing data to the graph gives it legitimacy. As a result, writes expose clearer authorization patterns that can be learned by a system such as IVD.

To better understand this, we consider a simple hypothetical OSN where people can create `friend` relationships, post pictures to their friends' profiles and see all pictures posted to their friends' profiles. Pictures are graph objects which hold the identifier of the user on whose profile the picture was posted and the `URL` to the content distribution network which stores the actual image file. A simplified implementation of the authorization logic for the *post* operation is shown in Algorithm 1. As IVD intercepts requests between the OSN and its graph database, it executes as a result of writing to the graph on line 3. Because this line is only executed when the logged-in user is a friend of the picture's target profile, i.e. the *pic.target* property, the rule "a picture can only be posted when its *target* property refers to a friend of the logged-in user" can be inferred.

```
   input : picture object pic
1  u = logged-in user;
2  if graph.associationExists(u, friend, pic.target) then
3  |   graph.write(pic)
```

Algorithm 1: A simplified implementation of the authorization checks required for picture posting in an OSN where users are only allowed to post on their friends' profiles.

```
   input : picture identifier pic_id
1  u = logged-in user;
2  pic = graph.getObject(pic_id);
3  if graph.associationExists(u, friend, pic.target) then
4  |   return pic
5  else
6  |   return nil
```

Algorithm 2: A simplified implementation of the authorization checks required for picture retrieval in an OSN where users are only allowed to see pictures posted on their friends' profiles.

On the other hand, when reading an image, the authorization checks happen after the graph reads, as shown in Algorithm 2. The algorithm is correct: the authorization checks only allow users to see pictures from their friends' profile. However, the *getObject* method—and as a result IVD—is called both when legitimate users access pictures from their friends' profile, and when prying users attempt to access pictures they do not have access to. Therefore, the authorization rule "a picture can only be read by a friend of the picture's *target* user" is not clear until line 4, outside IVD's scope.

In addition, OSNs' workloads are notoriously read-heavy, as newly created content is often broadcast to subscribers, followers or other connections, leading to several orders of magnitude more database reads than writes and to increased resource consumption for any system that has to inspect these operations. As a result, in this work we only focus on write operations and outline potential approaches for handling reads in future work (§VII). However, it is important to realize that authorization bugs in write operations can lead to data breaches, e.g. allowing an attacker to befriend arbitrary users without their consent would expose all data users shared only with their friends.

C. Facebook

Facebook is one of the biggest OSNs, having a correspondingly large graph database, both in terms of entities stored and

users accessing it. Its database contains more than one trillion entities, and receives over 10,000,000 peak writes per second.

To secure the data, most accesses to the graph database happen through a declarative privacy-aware framework. The framework implements rule-based authorization by allowing developers to associate with any entity type *authorization policies*, which are then automatically checked whenever an entity of that type is read or written. An authorization policy is a predicate that decides whether a user action should be allowed or not. A typical model represents them as ordered sets of *authorization rules*. Each rule can either allow, deny, or take no decision regarding the action. The rules are evaluated sequentially and the first decision taken is the overall result of the policy.

For example, the previously discussed feature of merging businesses can be implemented by creating a new business object and connecting each of the businesses to be merged with it through a MERGED_INTO association. Enforcing authorization checks is done by associating with the MERGED_INTO association type an authorization policy with a single rule that allows the creation of the association only if the two business objects being connected belong to the logged-in user.

We believe that while authorization policies are a very powerful tool, they require a great amount of engineering discipline. First, the policies must be complete; developers must reason about all possible cases, implement the appropriate rules and connect them in the appropriate order. Over time, this often leads to policies that are complex, hard to debug and hard to reason about in the first place. Second, the authorization policies must be checked on every database read or write; any database access that, for historical or engineering reasons, does not go through the privacy framework must manually enforce the correct authorization checks, which can easily be overlooked in a very large codebase modified by hundreds or thousands of engineers. This problem is compounded when multiple endpoints implement the same functionality for different platforms, e.g. a regular web interface, a mobile web interface and a REST API. Any authorization checks added to one of these endpoints must be replicated in all the others.

Even with structured authorization policies in place, bugs still creep in. Since its launch, Facebook's bug bounty program[1] received more than 2,400 valid submissions and awarded more than $4.3 million to more than 800 researchers around the world, with reports about business logic bugs becoming more common [12]. In Section V we present several case studies where IVD has or could have prevented exploits.

IVD works alongside authorization policies. While we advocate for writing correct policies and thoroughly enforcing them, we recognize that bugs are inevitable and we add a second layer of defense through IVD. All bugs detected and blocked by IVD should shortly after materialize in new authorization rules. As an added benefit, a missing authorization check that IVD detects in one specific endpoint can point to a systemic problem, whose fix will affect overall security. In this sense, IVD picks up where testing leaves off; while more testing may find more bugs, it is difficult to know if one has tested "enough" or more testing is needed. IVD mitigates this problem by contributing to a diversified set of bug finding approaches, whose combined strengths increase overall security.

[1]Facebook's bug bounty program encourages security researchers and whitehat hackers to poke at Facebook's systems, discover bugs and earn monetary rewards in exchange of disclosing them responsibly without accessing or mutating actual user data.

D. Threat Model

IVD protects against attacks which rely on improper or incomplete authorization checks, usually mounted through a publicly accessible web interface or API. We adopt a realistic threat model, where the attackers are either logged-out or have regular user accounts. They can make any number of requests and can pass arbitrary arguments to any endpoint exposed by the OSN but cannot modify the OSN's server-side code or otherwise interfere with its execution.

In the most basic form, the exploits involve passing identifiers of objects that are not under the attacker's control, in the hope that the OSN's code will not make appropriate authorization checks, and inadvertently mutate the objects. Such attacks are successful when developers miss a check, make incorrect assumptions regarding the data that their code processes, or rely on client-side code to perform authorization.

IVD only protects against attack vectors which involve unauthorized database requests made through usual APIs. While other web application attacks such as XSS, CSRF, denial of service, social engineering, and infrastructure compromise are important, their prevention and mitigation require significantly different approaches such as taint tracking, rate limiting or intrusion detection. These approaches are complementary to the enforcement of correct authorization checks.

III. Design

In theory, invariants could be arbitrary *graph predicates* expressed in a graph database query language [13]. For example, the following Lorel [14] query finds entities X and Y such that the same sequence of edge properties that connects X and Y connects *John* and Y:

```
select X, Y
from   Winners.author A, Winners.author X,
       A.#@P.Y, X.#@Q.Y
where  A.name = 'John'
and    path-of(P) = path-of(Q)
```

where # denotes a path of any length, *@P* binds the path to variable *P*, and *path-of* returns a sequence of edge properties. It can be easily seen that such a query would be prohibitively slow to perform in near-real time for all requests, at the scale of a large graph database.

To achieve scalability IVD uses a lightweight invariant design. The main insight behind our approach is that limiting the scope of the query still offers enough expressive power to catch many real-world problems. Our design drastically limits both the extent of the graph that can be accessed and

the predicates that can be used. An *object invariant* can only reference properties of the object being manipulated, while an *association invariant* can only reference properties of the association and of the two objects the association connects. We collectively call these properties *local properties*.

To mitigate the locality of this invariant design we introduce a small number of domain-specific *global properties* that can be referenced in addition to local properties, such as the identity of the currently logged-in user or the set of groups the logged-in user is an administrator of. This helps in two ways: first, being able to reference the identity of the logged-in user immediately allows creating more expressive invariants. Second, being able to directly reference objects that often determine a user's permission to perform an action (e.g. being an administrator of a group allows one to add other users to the group) allows IVD to essentially side-step the locality restriction for a small set of authorization-relevant objects. To provide the most benefit, the global properties have to be relevant to authorization checks and efficiently computable, i.e. involve traversing a small number of edges from known graph objects. Such properties are the logged-in user's connections or friends, or the business profiles that they administer.

Local or global properties are connected in an invariant predicate by invariant operators. The only two operators an IVD invariant can use are *property equality* and *association existence*. Equality predicates assert that two properties are always equal. For example

$$o1.property1 = o2.property2.subproperty$$

While object properties can have scalar or aggregate types, we only consider equality of scalar values and recursively enumerate the values in any aggregate property, such as the *o2.property2* dictionary in the previous example.

Association existence predicates assert that an association always exists between two objects. For example

$$\textit{logged-in user} \bigoplus_{\text{ATYPE}} o.propertyx$$

distills the constraint that a graph association of type ATYPE must exist between the logged-in user and the object referenced by the property *propertyx* of the object involved in the database operation. While association existence predicates and global properties have similarities, they have different use cases: the former are domain-agnostic, but require a significant amount of database queries to determine, while the latter are intended to be used for efficiently-computable sets of values that are relevant to authorization. As we detail later on (§IV-B), our implementation uses different approaches to infer each type of predicates.

Figure 2 shows the invariants that apply in the previously discussed scenario of merging several businesses into a single entity. The implementation creates a new business object and connects all previous business with it (for simplicity we only depict one connection) through an association of type MERGED_INTO. The creation of this association must only be permitted when the logged-in user is the owner of both businesses being connected, a constraint which is distilled in three invariants that respect the semantic restrictions imposed by IVD.

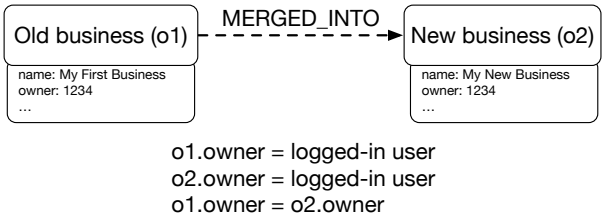


Fig. 2: Three invariants that apply when a user merges a business profile into another business profile. Adding the MERGED_INTO association must only be allowed when the logged-in user (a global property) is the owner of both businesses (local properties).

As this example shows, IVD invariants contain, in addition to a predicate, the context in which the predicate applies. Rather than create global invariants that would be checked on every operation, IVD uses a finer granularity and binds predicates to specific operations. For example, it associates a set of predicates to the operation of merging business profiles, a different set of predicates to the operation of creating a photo, and yet a different set to the operation of creating a comment. At runtime, only the predicates relevant to the current operation—if any—need to be checked. As a further improvement, we also associate predicates to the code which generated the request, i.e. a canonical representation of the URI or API endpoint that was used to initiate the request. This handles situations where an endpoint makes additional checks before issuing the request, e.g. the administrator area of an OSN could authorize users based on their IP address, hence can afford making requests that break invariants associated with openly accessible endpoints.

More formally, we define an *invariant category* as a 3-tuple (ENDPOINT, OTYPE, OPERATION) for object invariants, and a 5-tuple (ENDPOINT, O1TYPE, ATYPE, O2TYPE, OPERATION) for association invariants, where ENDPOINT is the source of the request, OTYPE, O1TYPE, O2TYPE and ATYPE are the types of the database entities involved in the request, and OPERATION is one of CREATE, DELETE or MUTATE.

Using this notation, we define an invariant as a pair

$$\mathcal{I} = (invariant\ category, \mathcal{P})$$

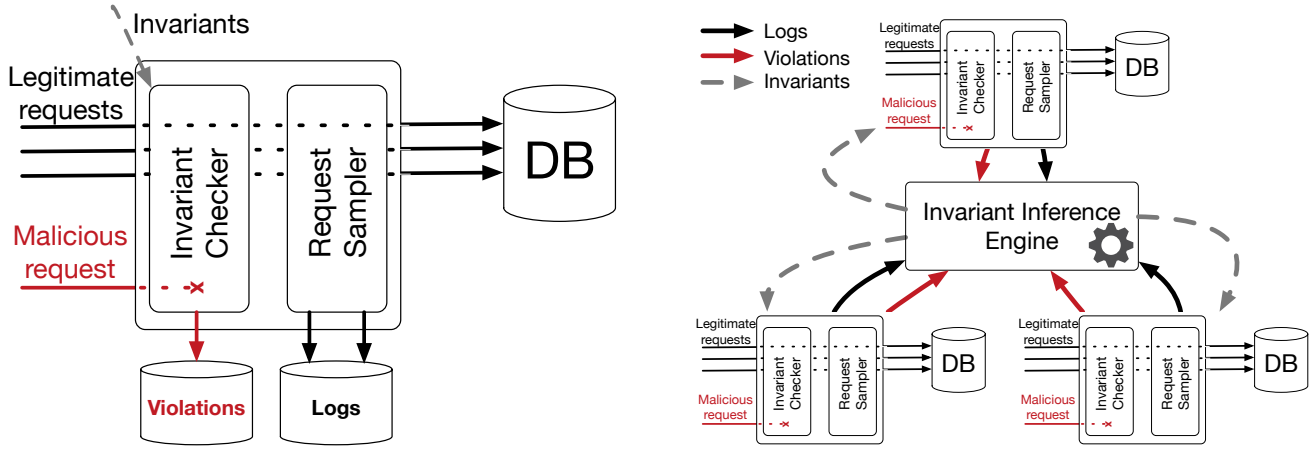
where

$$\mathcal{P}\colon properties \times properties \to \{\text{true}, \text{false}\}$$

is the invariant predicate function, restricted to the two operators previously discussed.

IVD's invariants fall under two broad classes: *authorization invariants* and *data validation invariants*.

Authorization invariants are the main focus of this work. They are constraints that involve a user's identity, e.g. a group post can only be created if the request comes from a member of



(a) INVARIANT DETECTOR's request sampler and invariant checker intercept requests between a client and a database system.

(b) INVARIANT DETECTOR uses a central invariant inference engine and distributed request samplers and invariant checkers.

Fig. 3: INVARIANT DETECTOR has three loosely coupled components: the *request sampler* logs a representative part of database requests, which are used for offline learning by the *invariant inference engine*. The learning process produces a set of likely invariants, which are passed to the *invariant checker* to be tested by evaluating them against all requests for a period of time. All invariants that are never broken are then *ratified* and used to block requests that do not match them.

the group. Breaking authorization invariants can have serious consequences, such as allowing an attacker to perform actions on behalf of a different user. In addition, such attacks can be difficult to spot and recover from, since the database remains consistent.

Conversely, data validation invariants are constraints that apply to the entire database, regardless of user identity. In other words, the database is consistent if and only if all data validation invariants hold. For example, an advertiser cannot remove their primary payment method if they have active advertisement campaigns, or the length of a post must be under 1000 characters. While IVD can catch data consistency bugs, as we will show in one case study (§V-C), consistency checks often require other predicates and are not the focus of our work.

In the rest of this section we discuss the three components that make up IVD, shown in Figure 3: the request sampler (§III-A), the invariant inference engine (§III-B) and the invariant checker (§III-C).

A. The Request Sampler

To achieve scalability, IVD does not attempt to infer invariants in real time. Instead, we gather representative data and use an offline learning process to mine for invariants. The request sampler is responsible for the data gathering step. Its purpose is to log a configurable number of requests from each invariant category, along with the values of local and global properties at the time of the database request.

To have access to all relevant data, the request sampler lives at the boundary between a client and a database server, as shown in Figure 3a. It intercepts database requests and has access to both local properties through the database request arguments and to global properties through a lightweight web application API.

When multiple clients and database instances are involved, the request sampler becomes a distributed system, sampler instances being colocated with either the clients or the databases to achieve good performance. For simplicity and robustness, the individual samplers are stateless and completely independent of each other. However, they still need to synchronize to globally log the desired number of requests from each invariant category. We solve this problem by globally assigning a *sampling rate* to each invariant category, which the samplers use as a probability to log. Due to the dynamic nature of the workloads seen by IVD, we can not use a static sampling rate. Instead an external component (not pictured in Figure 3) periodically analyses all logs and increases the rate for invariant categories that were under-sampled and decreases it for categories that were over-sampled. The new sampling rates are then distributed to every request sampler.

It is important to find the right balance in the number of requests to sample. A number too low leads to increased false positives while a number too high will be expensive in terms of storage and processing time of the logged data. We empirically found that 2000 samples per invariant category, resulting in an invariant inference engine workload of roughly 500 million total samples per learning cycle, matches well our computational capacity (§V-D).

B. The Invariant Inference Engine

The invariant inference engine looks for patterns in the data logged by the request sampler. It first splits log entries according to their invariant category. Each category is then analyzed separately, allowing for a large degree of parallelism.

To mitigate the false positives caused by sampling, we use a two-stage invariant deployment process. All newly-created invariants go through an *evaluation period*, in which they are checked against all requests. However, any requests that violate newly created invariants are not blocked, but rather cause the invariants to be invalidated. If an invariant is not violated during the evaluation period, it is *ratified* and any subsequent requests violating it are classified as malicious and blocked from reaching the database. The evaluation period mechanism is implemented in the centralized invariant inference engine rather than in the distributed invariant checkers, to allow for a simple and robust implementation of the latter. The invariant checkers only need to log all violations along with the invariant that caused them. The inference engine then picks up the violation logs and decides which invariants should be invalidated.

The inference engine executes periodically. At each run it analyzes the logs created since the previous run to find invariants to be put into evaluation mode, and the logs for the current evaluation period, i.e. the past five days in our implementation, to ratify invariants that passed evaluation. Our approach is stateless in that it does not look at the existing invariants, but only at the request and violation logs. This makes the algorithm easy to reason about and has the added benefit of making the inference process oblivious to transient failures in previous runs.

The invariant inference engine uses observations as ground truth, therefore it fundamentally needs to observe the OSN during normal operation. In particular, invariants for new OSN features can only be learned if authorization bugs in the new features are not actively exploited during the learning period. We consider this to be only a small limitation because new features usually go through testing and internal dogfooding where triggered bugs are expected to be reported. Regression bugs, on the other hand, are caught by virtue of preexisting invariants, and their detection does not depend on a period of quiescence.

C. The Invariant Checker

The invariant checker lives at the boundary between a client and the database system, similarly to the request sampler. However, unlike the request sampler, the checker runs synchronously on all database requests. For each database request, it first retrieves the endpoint that made the request and the involved entity types to determine the invariant category for the request. It then uses the category to get all relevant invariant predicates. The predicates are evaluated and any violations are logged. Furthermore, if a ratified invariant is violated, the database request gets aborted and an application exception is thrown.

Aborted requests cause notifications that trigger a manual investigation. An engineer can either confirm that the root cause of the violation is a bug and proceed to fix it, or deem the violated invariant spurious or no longer relevant. For the latter case, she will *blacklist* the invariant for the specific invariant category where the violation was triggered.

Most blacklisted invariants fall under one of three classes: coincidental correlations, modified product behavior, and rarely used features. The first involve conditions that had occurred almost always without being necessary for correct product behavior. For example, a user would have almost always seen a post before liking it (the *has seen* property is usually encoded as an association from the user to the post, which can be inferred into a association existence invariant), but this is not a requirement. The second class includes invariants that were valid for a previous OSN version but are not anymore, e.g. after switching from a policy where only a business profile's administrator is allowed to ask users to follow the business profile to a policy where users who already follow the business profile can invite their friends to do the same. Finally, some invariants are not correct but are learned because the code paths that cause their violation were never exercised during the learning period.

While spurious invariants are inevitable since our learning process bases ground truth on a limited number of observations, we have several defenses against them. First, the ratification algorithm requires the invariant to hold for a set minimum number of requests over at least five days before it can be enforced. This does not, however, protect against invariants that are no longer correct as a consequence of changes in system behavior. These situations are mitigated in two ways. First, invariants are also enforced on developer machines. This allows developers to notice problems early and remove the invariants. Second, code changes are canaried before being sent to the entire fleet of servers. The canaries can detect an abnormal number of failures and block the deployment until the situation is manually remediated. Finally, invariants can be manually blacklisted, with changes taking effect in a matter of seconds across the entire fleet of servers, as we will discuss in IV-C.

IV. Implementation

In this section we describe Invariant Detector's implementation, focusing on the challenges that we had to overcome to handle the scale of Facebook's workload. We separate the discussion into two: we first look at the request sampler and invariant checker, which are implemented in the database clients, and then discuss the invariant inference engine, which is built on top of Facebook's data analytics infrastructure.

A. Database Client Components

IVD's request sampler and invariant checker are implemented in Facebook's graph database clients, i.e. its web servers. The main reason for this placement is to avoid having to pass client state, e.g. the logged-in user, to the databases. This also allows us to distribute the load across many machines at the expense of having to make the invariants accessible on each of them. While Facebook offers several interfaces through which users can interact with it—desktop and mobile websites, APIs for external clients, mobile applications, and internal tools— they all share the same database API, which

```
input : Request sampler logs for the last day
output: invariant_category → invariants map
1  foreach invariant_category c in input do
       // check that we have enough requests
2      rqs = size(c.requests);
3      if rqs < EvalThreshold then continue;
4      all_eq_pairs = [];
5      foreach request rq in c.requests do
6          foreach p in rq.equal_properties do
7              all_eq_pairs.addAll(combinations(p, 2))
8          end
9      end
10     foreach property_pair in all_eq_pairs do
11         if all_eq_pairs.count(property_pair) == rqs then
               output[c].add(property_pair);
12     end
13 end
```

Algorithm 3: Pseudocode algorithm describing IVD's HiveQL invariant inference data pipeline for equality predicates.

```
input : Request sampler logs for the last 7 days,
        violation logs for the last month
output: invariant_category → invariants map
1  foreach invariant_category c in input do
2      checked = map();
3      foreach request rq in c.requests do
4          foreach iv in rq.checked_invariants do
5              if iv not in violations[c] then
                   checked[iv].add(rq.date);
6          end
7      end
8      foreach iv in checked do
9          if MeetsRatificationThreshold(checked[iv]) then
               output[c].add(iv);
10     end
11 end
```

Algorithm 4: Pseudocode algorithm describing IVD's HiveQL invariant ratification data pipeline.

conveniently supports the visitor pattern, making it easy to add new code to examine each database request.

The logged information for a request includes the invariant category, a mapping from values to sets of properties (local or global) that had those values, and the invariants that were checked when performing this request. The checked invariants are later used to determine how many times an invariant has been evaluated and decide whether it is ready to be ratified. For efficiency purposes, we do not attempt to infer potential new invariants at this point but instead defer the task to the invariant inference engine.

While the invariant category and the local properties are directly available in the request, the request sampler may have to query the graph database to retrieve global properties. However, because the actual logging is performed after responding to the web request, the request sampler does not introduce any user-noticeable delay.

The invariant checker's main implementation decision regards the storage of its invariants. Since the invariants are checked at every database write, with any delays being perceived as slower database replies, efficiency is critical. Our implementation keeps a copy of the invariants in each web server's memory. When this is not feasible or economical, a distributed in-memory store such as Memcached can be used as an alternative.

B. Invariant Inference Engine

The invariant inference engine is built on top of Facebook's data warehousing and analytics infrastructure [15], which mainly consists of three open-source systems: Scribe, responsible for collecting and aggregating the request sampler and invariant violation logs, Apache Hadoop [16], responsible for storing them, and Apache Hive [17], responsible for querying the data. The inference engine's embodiment is a set of HiveQL queries and Python scripts that define data pipelines for transforming (a) the raw logs into invariants, and (b) invariants under evaluation into ratified invariants.

The main challenge in designing the invariant inference engine was to formulate its algorithms in a fashion suited to HiveQL's data manipulation statements. The pseudocode for the algorithm is shown in Algorithm 3. The code first splits the log data by invariant category (line 1) and for each category, concurrently, computes the equality invariants by finding the sets of local and global properties that were always equal. For this, it first checks that sufficient requests are available to confidently create invariants (line 2). If the condition is met, it iterates through all requests, and for each of them iterates through its set of equal properties (lines 5–7). *rq.equal_properties* is itself a set of sets; each of its elements contains two or more properties that had the same value when the database operation occurred. Since some of these elements may form an invariant while others may only appear incidentally, all possible pairs of elements are computed and added to the *all_eq_pairs* list (line 6). Finally, the pairs which appeared in every request are output (line 10).

Invariants that use the association existence predicate are more challenging to infer. The logged information from the request sampler does not contain all the graph associations that exist between identifiers from the logged data because obtaining this information in real time can introduce substantial overhead in database requests. Therefore, the inference engine first obtains the relationships between the user logged-in at the time of the database request and the objects involved in the request by querying the graph database *post-hoc*. Association invariants are then created when an association of the same type exists for all log entries corresponding to an invariant category. While this offline analysis is not precise due to changes that may have happened to the graph since the logs were collected, it can only lead to missed invariants, and not false positives, thanks to the evaluation period that the invariants go through.

The invariant ratification data pipeline takes all invariants that were sufficiently evaluated—both in terms of number of days and number and requests—without ever being violated and marks them as ready to be enforced. Algorithm 4 describes the invariant ratification process. All invariant categories are processed concurrently; for each category and each request associated with it (line 1), the algorithm adds the invariants that were checked to the map *checked*, along with the date on which the request occurred (lines 4–6), to be later used by the ratification check (lines 8–11). Invariants that caused violations in any request for the current invariant category are ignored (line 5). Having all the invariants that always held and the the number of times they were evaluated, *MeetsRatificationThreshold* decides whether the invariant should be ratified (lines 9–11). The ratification requires a minimum of 500 requests evaluated every day for at least five of the past seven days. In addition, we require a minimum number of distinct values to be taken by the properties involved in the invariant, as we describe in more detail in §IV-D.

The inference engine is designed to be autonomous and resilient to workload changes and transient failures. The execution of the data pipelines is managed by an internal framework which takes a job's specification and automatically provides scheduling, monitoring, alerting, and simple reliability features such as automatic retries. Hive allows scaling the amount of data that we process through its MapReduce model by allocating more mappers or reducers, as needed, based on the amount of data collected. Finally, the pipelines are designed such that each execution is independent of previous executions, mitigating potential cascading failures. By virtue of this design, the code automatically picks up new invariant categories and infers new invariants when new features are added to Facebook, and purges invariants if the features they were associated with are no longer used.

In addition to the daily invariant computations, we perform hourly sampling rate adjustments. To determine the sampling rate, we estimate the number of requests that will be made for each invariant category by looking at the category's history. First, the request logger writes to each log entry the current sampling rate for its invariant category. We then approximate for each invariant category the total number of requests that happened during period T as

$$N \approx \sum_{L \ logged \ during \ T} 1/R_L$$

where R_L is the sampling rate attached to log entry L. There are different options for choosing the period on which to base the estimation: the previous hour, the same time on the previous day or the same time the previous week. To be able to adjust quickly to new traffic, and because our workload is seldom bursty, we use the previous three hours, which in our experience results in daily counts close to our targeted number of samples.

C. Data Distribution

IVD needs to transfer large amounts of data between its central inference engine and the distributed components: logs from the request sampler and invariant enforcer, invariants to the invariant enforcer, and sampling rates to the request sampler. We use separate mechanisms for each of them.

For transferring logs we leverage Scribe [15], Facebook's dedicated logging infrastructure. Scribe offers an API to describe the data types to be logged, and automatically creates the needed boilerplate code and initializes the Hadoop data store. We refer the interested reader to the original paper for further details.

Distributing the invariants is done by leveraging Facebook's quasi-continuous deployment model. Our implementation piggybacks on the infrastructure that sends application updates to the web servers. Besides code, the updates also include new web server cache data to be loaded on server restart, which is where we bundle our invariants. The frequency of updates is higher than the daily invariant computation, making this distribution model effective for our use case.

Coupling application updates with invariant updates has the added benefit of leveraging the canarying process used for testing web application code to also test the code's interaction with newly-ratified invariants. This protects against deploying invariants that would significantly affect Facebook's operation, either because of bugs in IVD or due to adverse interaction that were not triggered during previous testing.

Even if incorrect invariants are deployed, they can be rapidly blacklisted. Configerator [18], a system built on top of Zookeeper [19], offers propagation delays of seconds, which allow prompt reaction in the event an incorrect invariant is detected in production systems. Conversely, this system also allows manually adding new invariants to be enforced, which can be an effective first-line mitigation for bugs found through other means, since writing and deploying an invariant is significantly faster than writing and deploying new code. In addition, we use Configerator to distribute sampling rates.

D. Optimizations and Heuristics

Post-send processing. In order to minimize perceived database response times, we keep a large part of IVD's code off the critical path. We do this by leveraging web server functionality that allows registered callbacks to be executed asynchronously after the HTTP response has been sent to the user. All request sampler code executes in a post-send processing context.

Enforcement excuses. We improve IVD's precision by manually specifying domain-specific rules that *excuse* violations, i.e. allow requests to proceed even though they have violated an invariant. As the excuses are checked just before a request is about to be blocked, they have access to more information than was available when the invariant was created, in particular to the values that do not satisfy the invariant and much of the application state. In addition, the excuses are rarely executed since violations of ratified invariants are relatively rare, affording them more thoroughness. IVD currently uses 17

enforcement excuses that target common classes of false positives, as we discuss in more detail in §V-C. Due to their domain-specific nature, the enforcement excuses may vary between OSNs, depending on the complexity of their underlying data model. We believe Facebook has a relatively complex model and other implementations will require fewer enforcement excuses.

Distinct value count. The ratification conditions (§IV-B) include a minimum threshold for the number of distinct values that the properties involved in the invariant have taken. This avoids ratifying incidental invariants that represent version numbers, image resolutions, timestamps, or that are currently unused, e.g. fields always set to "0" or the empty string. We empirically picked 1440 for the number of daily unique values required to ratify an invariant, i.e. one per minute. Since the values that we are interested in are authorization-related—often user identifiers—the minimum threshold of distinct values has the side effect of limiting invariant generation to features involving more than 1440 users every day. This number can be easily customized based on the size of the OSN and more weight can be given to user identifiers used during internal testing.

V. Evaluation

Invariant Detector's initial incarnation was deployed at Facebook more than two years ago and has since detected several critical vulnerabilities that have since been fixed. In this section we give an intuition on the amount of work IVD does, show details on the invariants that it infers and ratifies, describe our experience with running IVD at Facebook's scale, and evaluate its effectiveness and performance.

A. IVD *Deployment at Facebook*

We begin with an overall picture of Facebook's IVD setup to give the reader a better sense of scale. IVD checks more than 10,000,000 peak database write requests per second and uses a sample of roughly 500 million requests for the daily invariant inference engine execution. At the time of our evaluation, the inference engine produced 226,598 invariants that that were put into evaluation mode, out of which 158,205 were ratified at the end of the evaluation period.

In the following we look in more detail at the invariants to understand (1) which predicates are most often inferred, (2) which invariants fail evaluation and (3) which invariants are eventually enforced. To present representative examples, we rank the invariant predicates according to the number of invariant categories they are associated with. Intuitively, a predicate associated with more invariant categories applies more broadly to changes that are made to the social graph.

Table I shows the top 10 predicates put into evaluation, the number of invariant categories for which they have been put into evaluation, and the number of categories for which they were ratified. For example, the first line can be read as: there are 40,150 invariant categories, i.e. (ENDPOINT, OPERATION, ATYPE, O1TYPE, O2TYPE) tuples, for which the predicate "the association's first object was the logged-in user" held

Invariant Predicate	**#Evaluated Invariants**	**#Enforced Invariants**
logged-in user = o1	40,150	27,144
logged-in user = o1.creator_id	9,018	6,738
logged-in user = o2	8,257	5,781
logged-in user = o.owner_id	6,250	5,621
logged-in user = o2.creator_id	6,146	4,691
o1 = o2	6,046	4,695
logged-in user = o2.owner_id	5,834	4,106
logged-in user = o1.owner_id	4,716	3,404
o1 = o2.owner_id	4,210	3,112
the logged-in user is an administrator of Page o1	3,951	2,590
... 2192 more		
Total	**226,598**	**158,205**

TABLE I: Top 10 predicates by number of invariant categories they are associated with. The numbers include both objects and associations writes.

in all sampled requests whenever ENDPOINT performed OPERATION on an association of type ATYPE between objects of types O1TYPE and O2TYPE. Furthermore, the predicate passed evaluation for 27,144 of the 40,150 invariant categories. As it can be seen, all the top predicates reference common privacy-related property names, which match against the logged-in user (USER) or one of the objects involved in the request, i.e. O1 or O2. These properties are common to many object types, resulting in them being referenced in many invariant categories. By contrast, predicates which appear in fewer categories reference more specific properties, e.g. `o1.user = o2.job_data.owner_id`.

The invariants which did not pass evaluation can be put into two classes. First, there are invariants which did not hold when tested against 100% of production requests. This reason accounts for 30,098 invariants. Second, there are invariants which always held but were not involved in a sufficiently large and diverse number of requests for our system to have confidence in their correctness, as described in §IV-D. This accounts for 55,557 invariants. Note that the same invariant predicate may cause violations for one particular invariant category, not meet ratification threshold for another, and be ratified for yet another.

The predicate that caused the most violations in our evaluation step is *o2 is a friend of the logged-in user*. Intuitively, this happens because users perform most of their interactions with friends, resulting in the invariant being picked up initially. However, many interactions are also valid for non-friends, resulting in the invariant failing evaluation. A related example is the predicate *the logged-in user recently communicated with o2*. While that predicate usually holds, it is only an incidental relationship and is not required. We see similar patterns in all top 10 invariants ranked by number of violations caused in the evaluation stage. We therefore conclude that checking the invariants against all production requests, rather than against a sample, is necessary for having a set of correct invariants.

The invariants which did not meet the ratification volume requirements either reference properties that take a limited set of values or are related to features that are only available to

a limited number of users, either by virtue of their nature or because they are not fully released. For example, certain object properties contain enumerated values or version numbers. Not only do these usually not concern privacy, but they can also change over time, e.g. as a new version is released, leading to false positives. On the other hand, product features with limited exposure could benefit from a lower invariant ratification threshold. For example, fundraisers are only available for qualified US-based 501(c)(3) nonprofits, some business features are only available to the employees of companies that have come into an agreement with Facebook, and some product features are only available to Facebook employees. We are considering ways to enforce these invariants without impacting the system's overall false positive rate such as using a finer granularity for the invariant categories.

B. Effectiveness

IVD is a defense-in-depth system, therefore we expect it to only jump into action on the rare occasions when authorization checks are missing. However, even before that, we have the possibility of quantifying its effectiveness by looking in more detail at the invariants that it creates.

We define *authorization coverage* as the fraction of authorization checks required for a system's correct functioning that are actually performed by its implementation. A correct system has therefore 100% authorization coverage. While ideally we would compute IVD's authorization coverage by comparing its invariants against Facebook's specification, this is not possible due to the absence of a formal specification, which is very often the case in large-scale projects. Rather than evaluate IVD's effectiveness on a smaller toy OSN, we believe we can obtain more insights into the system by performing tractable measurements that can be used as a proxy for its actual authorization coverage.

One way to estimate authorization coverage is to compute the fraction of invariant categories that appear in database writes, that have at least one invariant associated with them. This provides a rough overall estimate of the protection offered by IVD: it infers invariants for 50% of invariant categories, and enforces them for 36% of categories. These numbers, however, may not accurately represent authorization coverage since some categories do not perform privacy-sensitive operations, e.g. logging, while others may require multiple invariants, of which only some have been inferred.

To obtain a more granular view of IVD's authorization coverage, we use Facebook's code as its specification and manually compare IVD's invariants against a sample of the authorization checks implemented in Facebook's code. Besides helping us better quantify authorization coverage, this approach also gives us insights into whether the expressive power of IVD's invariants is adequate, and has the added benefit of comparing IVD invariants against checks that we can assume to be essential for the correct functioning of the OSN.

To achieve this, we manually went through Facebook's codebase, identified a sample of authorization checks, and verified whether equivalent invariants existed. Such invariants would prevent a hypothetical attempt to abuse Facebook in case developers missed the check. As described in §II-C, Facebook's data access framework offers a structured way of writing authorization policies. This allowed us to identify the authorization rules by simply looking for classes that use the `PermissionsValidator` trait. Out of roughly 1,000 such classes, we manually inspected the 22 which deny access if the logged-in user does not satisfy a condition. We pick these because (a) breaking them would have a significant impact, (b) they are a prime target of IVD by design, and (c) we could easily identify them by matching the class name against the regular expression `DenyIfViewerIs.*Not.*`, e.g. `DenyIfViewerIsNotOwnerOfPage`.

For each rule, we inspected its code to determine the object and association types that it applies to, and the constraint that it enforces. We then looked at the invariants IVD created for those types and checked whether any of them corresponded to the coded condition. We found that out of the 22 authorization rules, 7 are unused (either deprecated or part of an upcoming feature), and one is only enforced on reads. Out of the remaining 14 rules, 10 have a corresponding invariant and 4 do not. It is important to note that IVD has also inferred authorization rules not present in the codebase, as we will describe in more detail in §V-C.

While it is difficult to draw definitive conclusions from these observations, we consider them a good indication that IVD can adequately represent and learn authorization rules. To further confirm this we looked at IVD's bottom line contribution, expressed as the number of violations engineers have acted upon. More precisely, we looked at a period of 6 months and found 23 reports (out of a total of 222 IVD reports) which had one or more code changes associated with them and moreover, we manually confirmed that the changes are related to authorization checks. To put this into perspective, Facebook's bug bounty program resulted in 526 valid submissions (out of a total of 13,233 submissions) in a 12 months period [12]. Because Facebook's bug bounty program accepts all types of security issues across all of Facebook's services [20], bug bounty submissions are not limited to authorization bugs nor to Facebook proper. We therefore consider IVD's results to provide an important contribution to Facebook's security.

We next look at several specific bug instances to better understand IVD's strengths and blind spots.

C. Case Studies

IVD blocked abnormal behavior in several different scenarios. Some involved benign users exercising a Facebook feature in a way that was not expected or not taken into account by the engineers. Others, however, were security researchers participating in the bug bounty program or malicious actors attempting to discover and exploit vulnerabilities in Facebook. We present in more detail several cases where IVD stopped bugs from being exploited and several cases where it did not, along with the lessons that we have learned.

True Positives. As described in §I, a lesser known Facebook feature allows its users to merge business Pages that they own. During the merge process, likes, followers and reviews are consolidated into a single Page. Internally, this works by creating a *Page merge* object and connecting all Pages involved in the merge to it. IVD correctly inferred that when connecting a Page object to a Page merge object, the logged-in user must have an owner association with the Page. This invariant was broken when a security researcher attempted to exploit the merge system by crafting a merge request containing identifiers of Pages that were not under their control. Even though the researcher could not exploit the bug by virtue of IVD, Facebook issued them a bounty for triggering it.

Another Facebook bug could have allowed any business Page administrator to list other arbitrary Pages as their business' *clients* through a maliciously crafted request. The bug was caused by a missing authorization check that IVD correctly inferred: an administrator association must exist between the logged-in user and the Page being connected for the operation to be performed.

Another Facebook feature allows users to transform a personal profile into a business Page. Users who inadvertently create a personal profile to represent a business use this feature to convert the profile and benefit from additional business features. The conversion process maintains all content and connections with other users. Internally, this process (1) creates a Page object, and (2), copies all content and connections from the existing user profile to the newly-created Page. One of the endpoints implementing this feature accepted a Page object identifier and a set of user profile identifiers to be added as followers to the Page. IVD correctly inferred that the logged in user must be friends with the account being added as a follower to the business. An attacker who attempted to exploit this vulnerability aimed to force a number of users to follow a Page they did not choose to.

While IVD aims towards authorization bugs, it inherently detects other unintended changes to an OSN. One example involved a refactoring that split the *group member* association into two: *confirmed member* and *unconfirmed member*. The now-legacy *member* relations was left into place to be used by code predating the refactoring, IVD included, that did not distinguish between the confirmed and unconfirmed status. The new and old associations were therefore supposed to be synchronized. However, as IVD pointed out soon after the change, there were rare circumstances where the member association was no longer created, while the corresponding new associations were, leading to inconsistencies in the graph. IVD exposed the inconsistencies, during legitimate user activity, as violations of invariants requiring the logged-in user to be member of a particular group.

False Negatives. An important part of IVD's development is the post-mortem analyses of uncaught vulnerabilities that affected Facebook. These allow us to understand IVD's blind spots, find implementation bugs in IVD itself, and significantly improve the system over time. In the next paragraphs we discuss several authorization bugs that affected Facebook during the course of our study and did not trigger invariant violations.

A bug not caught by IVD allowed users to delete arbitrary videos [21]. To exploit it, a malicious user would first (1) create a comment, (2), attach the target video to it, and (3), delete the comment. Step 2 caused the comment to take ownership of the video, while step 3 caused the comment deletion to also trigger the video's deletion. Neither step 2 nor 3 performed the access control checks necessary to prevent the problem. While Facebook's web interface did not allow arbitrary videos to be attached to comments, its REST API did.

An IVD invariant that would have caught the bug would have checked whether the video owner is the same as the comment author in step 2, when adding the association between the video and the comment. However, IVD was not enforcing this invariant at the time the bug was discovered because the vulnerable endpoint was part of a recent product feature that was tested extensively via the web interface before release but not significantly via Facebook's REST API. Therefore, the volume of sampled API requests was not sufficient to meet IVD's ratification criteria. We mitigated this problem through a more aggressive request sampling policy for test users and employees, which results in invariants being created faster after new features are introduced.

Another bug [22] allowed any user to change the cover photo of any event they could see. The root cause of this problem was a missing permission check in the code handling the cover photo update. IVD did not detect the problem because of the complex authorization policy that determines whether a user can edit an event cover photo. The policy recognizes three situations: the event is public and has no cover photo yet, the event is created for a group, or the event is created for a Page. Each case is treated separately and additional case-specific checks, such as whether the user is an administrator of the group owning the event, are performed. In particular, IVD failed to create an invariant because the permission check requires a disjunction predicate and IVD lacks such expressiveness (§III).

False Positives. False positives are invariant violations caused by legitimate activity. They are inevitable artifacts that result from basing ground truth on a limited number of observations. We use two approaches to handle them: we quickly stop further blocking by manually blacklisting invariants and subsequently target their underlying cause with enforcement excuses (§IV-D).

The most commonly-excused violations in our setup, amounting to 81% of all violations, are those that are not authorization-relevant, as described in §III. We heuristically detect whether a violation is authorization-relevant by looking at the runtime types of the objects involved in the graph request and at the name of the properties that the violated invariant references. The object types are compared against a whitelist containing the types known to be used in authorization checks, e.g. USER or PAGE, the association type is compared against a blacklist that holds types which are known to never be relevant

to authorization, e.g. logging, while the property names are matched against strings commonly used for authorization properties, e.g. *"owner"*, *"privacy"*. If none of the above checks indicates an authorization-related violation we default to excusing it. These violations do not cause requests to be blocked but they are nevertheless logged, allowing for further investigation.

Another enforcement excuse commonly triggered in our setup concerns requests involving different identifiers that refer to the same user. This can happen because a user might act as their personal account, their business account, their advertiser account or as a Page that they administer. These accounts are represented by separate, but connected graph objects. As violations are relatively rare events, we can afford to inspect the graph in more detail and verify whether the mismatching identifiers point to objects that are connected in a fashion that indicates they refer to the same person, and if so, we excuse the violations. This excuse applied to 17% of all violations.

The remaining excuses cover a smaller fraction of violations but they nevertheless make an important contribution towards keeping the number of users affected by spurious invariants and the number of reports that have to be manually investigated low.

For example, we noticed violations involving code that lazily performs actions, such as minor graph garbage collection, after handling a user's request. These actions appear to be performed by the user who originally made the request but they can operate on other users' data and cause spurious violations. Since the code relevant to these actions is localized, we use a call stack-based enforcement excuse that matches call stack frames to specific functions assumed to correctly perform maintenance activities to the graph.

Other excused scenarios include unowned Pages, memorialized users, and different users acting on an object belonging to a Page or group that they commonly administer. Overall, the excuses rule out approximately 99% of violations, with the remaining 1% having to be manually inspected. We believe we can further decrease this number by identifying and excusing more spurious types of violations. IVD's final false positive rate, i.e. the number of requests incorrectly blocked over the number of requests which had invariants associated with them is roughly 0.00000004%.

Vulnerability Mitigation. IVD may detect but not mitigate vulnerabilities. This behavior can occur in web application endpoints that manipulate multiple entity types. Consider the endpoint that creates the Page merge object, associations between the merge object and individual Page objects and associations between the merge object and users. IVD considers these operations to belong to different invariant categories (§III) and may infer and enforce different predicates on them. It is possible for an attacker to cause an invariant violation on the third step of the process but not earlier, when the sensitive merge object is created. If this happens in the absence of transactional semantics IVD blocks the last operation but allows all previous operations to complete, potentially resulting in merged, albeit corrupt, Page.

From the attacker's perspective the attack is successful so there is an incentive to keep exploiting it. IVD however reports a continuous stream of invariant violations as the attack unfolds and pinpoints the endpoint that is being exploited. IVD's behavior is still valuable despite the system not being able to automatically mitigate the vulnerability and potentially causing data corruption.

We conclude that IVD invariant violations must be monitored by engineers closely; the engineers should try to understand the expected product behavior and either whitelist the invariant or fix the vulnerable endpoint.

D. Performance

IVD sits in a performance-critical part of the software stack, between the web application and the graph database, therefore its performance impacts most application features. Since IVD is optimized to keep most of its data in memory and read infrequently from the social graph beyond what is readily available in the current request, its average execution can be approximated by a CPU-bound workload.

To understand IVD's performance impact we use a sampling profiler that analyzes all of Facebook's web servers and aggregates the data globally. At Facebook the overall performance and resource consumption of the web application are closely monitored and distilled in a metric named "app weight". Any changes that regress the app weight must be well-motivated. The measurements show that IVD's contribution to the total CPU time used executing web application code is a mere 0.014%.

To further quantify IVD's performance impact, we instrument its code to measure the additional wall time it adds to each database write and log a sample of the measurements. An analysis of 40,000 samples yields a median overhead of 0.1ms per database write, while the 99th percentile, i.e. the time taken by the check that is slower than 99% of all checks, is 5ms. However, these numbers are only an upper bound on the end-to-end time penalty that IVD adds because Facebook's code may perform multiple operations concurrently. In particular, unrelated I/O operations can be serviced while IVD executes, leading to an increased level of parallelism but not necessarily an increased end-to-end wall time.

To obtain a better idea of end-to-end wall time overhead we measured the time needed to serve requests with and without IVD enabled. We performed the analysis on Facebook's production systems in order to take into account all interactions that may not be visible in a testing environment and obtain a result representative of IVD's intended use. We logged the performance characteristics of approximately 1 million user requests that resulted in at least one database write. Since the request wall time is significantly affected by the request arguments, e.g. the time taken by a file upload request depends on file size and on the quality of the internet connection between the user and the OSN, we compared median times. We separately recorded the time needed to send a response back to the user and post-processing time—where IVD's request sampler runs—and observed a median difference

of 1 millisecond for the former and 2 milliseconds for the latter. We therefore conclude that IVD's performance impact is virtually imperceptible to an OSN's end-user.

The performance of IVD's invariant inference engine is also important. The daily inference engine runs need to compute new invariants in less than 24 hours. Our inference engine implementation typically needs 4–6 hours to compute new invariants, using a peak of 1,000 mappers and reducers.

VI. Related work

Traditionally, research on fighting web application bugs has focused on runtime data-flow tracking and static analysis. The former mainly targets injection vulnerabilities such as XSS and SQL injection [23], [24], [25], while the latter attempts to find, among others, missing authorization checks. Systems such as Fix Me [26], MiMoSa [27], RoleCast [28], SaferPHP [29], and WAPTEC [30] use static analysis to check a web applications's security logic. However, such analyses face a fundamental problem when confronted with applications that have complex access control policies: in the absence of a specification, it is impossible to decide whether an operation should be allowed, and indeed, this approach has been largely restricted to applications that use a simple privacy model, of regular and privileged users. IVD does not have this limitation.

One way to side-step this problem is to use the application's usual behavior as the specification. APP_LogGIC [31], Swaddler [32], Waler [2], and ZigZag [33] use Daikon [34] to extract invariants that are part of the "intended" application specification. The invariants are then either checked at runtime or used in a model checking step to identify paths on which they are not enforced. While their invariants are based on program state, we propose the simpler and scalable approach of basing them on database queries. In addition, we avoid the expensive model checking step and ensure that runtime checking of the invariants has virtually no overhead.

Other systems base their invariants on different data. BLOCK [5] generates invariants from web requests and responses to detect state violation attacks, while InteGuard [6] finds invariants in the HTTP traffic between providers, integrators, and end-users, with the goal of protecting integrators from malicious users. While these approaches have the advantage of working outside the application they protect, the expressive power of their invariants is inevitably lower as they have limited access to application state. In addition, the graph database layer affords us finer invariant granularity than the HTTP layer.

IVD also has similarities to anomaly detection systems [35], [36], [32]. While they share the same high-level idea, anomaly detection systems assign to each request an *anomaly score* which they compare against a threshold determined during the learning phase. The score is usually computed using statistical models such as string character distribution or token finder, which makes them suitable for protecting against attacks that involve specific input patterns, but less so against authorization bugs. Several anomaly detection systems have been proposed specifically to target SQL queries [37], [38], and proved to be effective against SQL injection or XSS.

```
   input : picture identifier pic_id
1  u = logged-in user;
2  auth_data = graph.getAuthInfo(pic_id);
3  if graph.associationExists(u, friend, auth_data.target)
   then
4  |  return graph.getObject(pic_id);
5  else
6  |  return nil
```

Algorithm 5: Possible approach for making IVD suitable for reads by having the graph database API return an object's authorization-relevant information separately.

```
   input : picture identifier pic_id
1  u = logged-in user;
2  pic = graph.getObject(pic_id);
3  if graph.associationExists(u, friend, pic.target) then
4  |  IVD.onAuthorizationSuccess(pic);
5  |  return pic;
6  else
7  |  return nil
```

Algorithm 6: Possible approach for making IVD suitable for reads by explicitly informing it of successful authorization checks. Line 4 assumes IVD is is a globally-accessible object.

Working to protect network applications from malicious users, Vigilante [39] uses attack signatures rather than invariants. The signatures are generated by instrumentation that looks for attacks that rely on detectable exploit mechanisms such as buffer overflows. However, this approach does not work in the context of semantic bugs, where a generic method of detecting an attack does not exist. Also using attack signatures, intrusion detection systems such as Snort [40] protect against known vulnerabilities. However, this technique has limited applicability in proprietary applications, such as the ones powering OSNs.

VII. Future Work

IVD's deployment at Facebook does not currently cover database reads for reasons we have laid out in §II-B. The key insight into making IVD applicable to reads is to inform it of successful authorization. We propose two approaches to achieve this:

1) Separately read authorization metadata and payload.
2) Explicitly inform IVD of successful authorization.

The former approach involves splitting object attributes into two categories: relevant to authorization and irrelevant to authorization. The database would then provide a separate API for reading only the authorization-relevant attributes. Algorithm 5 shows the application of this approach to Algorithm 2. IVD would be invoked, as before, during the *getObject* call, but not during the *getAuthorizationInfo* call.

The latter approach does not involve modifying the graph API but requires the authorization code to explicitly notify IVD

when authorization is successful. Algorithm 6 conceptually presents the idea, applied to the same algorithm as above. Similarly to the handling of writes, IVD has at line 4 the opportunity to learn or to block accesses to the object by throwing an exception. In practice, the *ivd.onAuthorizationSuccessfull* call can be made by the authorization framework without requiring changes to user code.

A different direction for future work is to mitigate IVD's reliance on an attack-free learning period (§III-B). A potential solution involves flagging for manual review invariants that were *almost* ratified, i.e. they were only broken by requests coming from a small group of users. Alternatively, the invariants could be directly ratified, but then automatically disabled if they block requests coming from a sufficiently large number of users. More complex ratification criteria can be borrowed from the field of robust statistics [41].

Finally, IVD can be extended to learn more complex invariants. The system currently protects against bugs caused by missing authorization checks that can be expressed as equality between object attributes, direct relationships, or a conjunctions thereof. Potential extensions could allow IVD to infer and enforce disjunctions, or introduce predicates that capture indirect relationships, i.e. inspect objects separated from the logged-in user by more than one edge in the graph.

VIII. Conclusion

We have presented IVD, a defense-in-depth system that protects online social networks against missing or incorrect authorization checks. IVD works by inferring invariants from graph data query patterns and, after a short evaluation period, blocks any requests in which the invariants do not hold.

IVD's main novelty rests in its focus on the highly interconnected data model specific to online social networks, which allows inferring meaningful invariants at the database layer, and in the design and implementation decisions that allow it to learn and enforce invariants at an unprecedented scale. To our knowledge, IVD is the first invariant detection system that checks hundreds of thousands of invariants against millions of requests every second, made to one of the largest graph databases in the world. Additionally, IVD tackles the inherent susceptibility of dynamic invariant detection systems to false positives through a two-step evaluation and ratification process, and a set of effective domain-specific enforcement excuses.

IVD does have limitations stemming from the trade-offs of our design: deciding to learn at the database layer allows finding authorization-relevant invariants but has limited applicability for read operations, and the restricted invariant format offers good performance but does not allow inferring all authorization checks that the code may perform. IVD is therefore not a replacement for good engineering practices, security audits or bug bounty programs. However we have found it to be an effective additional layer of defense at Facebook.

IX. Acknowledgements

We would like to thank Ben Mathews and Alec Muffett for their early work on IVD, and the engineers from Facebook's security and product teams who investigated invariant violations. We would also like to thank Pieter Hooimeijer, Christopher Palow, Steve Weiss and John Lyle for helping us shape the paper, and our shepherd, Nikita Borisov, and the anonymous reviewers for their invaluable feedback.

References

[1] "OWASP Top 10 2013," https://www.owasp.org/index.php/Top_10_2013-Introduction.

[2] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the USENIX Security Symposium*, vol. 58, 2010.

[3] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications." in *Proceedings of the USENIX Security Symposium*, 2011.

[4] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Bandsaw: Log-powered test scenario generation for distributed systems," in *SOSP Work In Progress*, Cascais, Portugal, 2011.

[5] X. Li and Y. Xue, "Block: a black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011.

[6] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations." in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[7] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[8] "FlockDB: A distributed, fault-tolerant graph database," https://github.com/twitter/flockdb.

[9] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX Annual Technical Conference*, 2013.

[10] "Neo4j decreases development time-to-market for LinkedIn's Chitu App," https://neo4j.com/case-studies/linkedin-china/?ref=solutions.

[11] S. Sakr, S. Elnikety, and Y. He, "G-SPARQL: A hybrid engine for querying large attributed graphs," in *ACM International Conference on Information and Knowledge Management*, 2012.

[12] "2015 Highlights: Less low-hanging fruit," https://www.facebook.com/notes/facebook-bug-bounty/2015-highlights-less-low-hanging-fruit/1225168744164016.

[13] P. T. Wood, "Query languages for graph databases," *ACM SIGMOD Record*, vol. 41, no. 1, Apr. 2012.

[14] S. Abiteboul, D. Quass, J. Mchugh, J. Widom, and J. Wiener, "The Lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, 1997.

[15] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, Aug. 2009.

[18] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at Facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the USENIX Annual Technical Conference*, 2010.

[20] "Facebook Whitehat," https://www.facebook.com/whitehat.

[21] "Facebook's bug - delete any video from Facebook," http://tinyurl.com/j29loqa.

[22] "Facebook whitehat information," http://tinyurl.com/zzvyumj.

[23] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.
[24] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the Network and Distributed System Security Symposium*, 2007.
[25] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.
[26] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications." in *Proceedings of the Network and Distributed System Security Symposium*, 2013.
[27] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.
[28] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," *ACM SIGPLAN Notices*, vol. 46, no. 10, 2011.
[29] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011.
[30] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.
[31] G. Stergiopoulos, B. Tsoumas, and D. Gritzalis, "Hunting application-level logical errors," in *Proceedings of the International Symposium on Engineering Secure Software and Systems*. Springer, 2012.
[32] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007.
[33] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Zigzag: Automatically hardening web applications against client-side validation vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2015.
[34] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, 2007.
[35] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.
[36] K. Leung and C. Leckie, "Unsupervised anomaly detection in network intrusion detection using clusters," in *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*. Australian Computer Society, Inc., 2005.
[37] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2005.
[38] S. Y. Lee, W. L. Low, and P. Y. Wong, "Learning fingerprints for a database intrusion detection system," in *European Symposium on Research in Computer Security*, 2002.
[39] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 2005.
[40] M. Roesch, "Snort - lightweight intrusion detection for networks," in *USENIX Conference on System Administration*, 1999.
[41] F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel, *Robust Statistics: The Approach Based on Influence Functions.* John Wiley and Sons, 1986.