Scaling Static Analyses at Facebook

Dino Distefano, Manuel Fähndrich, Francesco Logozzo and Peter W. O'Hearn
Facebook

1. Introduction

Static analysis tools are programs that examine, and attempt to draw conclusions about, the source of other programs, without running them. At Facebook we have been investing in advanced static analysis tools that employ reasoning techniques similar to those from program verification. The tools we describe (Infer and Zoncolan) target issues related to crashes and to the security of our services, they perform sometimes complex reasoning spanning many procedures or files, and they are integrated into engineering workflows in a way that attempts to bring value while minimizing friction. They run on code modifications, participating as bots during the code review process. Infer targets our mobile apps as well as our backend C++ code, codebases with 10s of millions of lines; it has seen over 100 thousand reported issues fixed by developers before code reaches production. Zoncolan targets the 100 million lines of Hack code, and is additionally integrated in the workflow used by security engineers; it has led to thousands of fixes of security and privacy bugs, outperforming any other detection method used at Facebook for such vulnerabilities. We describe the human and technical challenges encountered and lessons we have learned in developing and deploying these analyses.

There has been a tremendous amount of work on static analysis, both in industry and academia, and we will not attempt to survey that material here. Rather, we present our rationale for, and results from, using techniques similar to ones that might be encountered at the edge of the research literature, not only simple techniques which are much easier to make scale. We intend that this should complement other reports on industrial static analysis and formal methods (e.g., [17, 6, 1, 13]), and hope that such perspectives can provide input both to future research and to further industrial use of static analysis.

We continue in the next section by discussing the three dimensions (bugs that matter, people, and actioned/missed bugs) that drive our work. The rest of the paper describes our experience developing and deploying the analyses, their impact, and the techniques that underpin our tools.

2. Context for Static Analysis at Facebook

Bugs that Matter We use static analysis to prevent bugs that would affect our products, and we rely on our engineers' judgement as well as data from production to tell us the bugs that matter the most.

It is important for a static analysis developer to realize that not all bugs are the same: different bugs can have different levels of importance or severity depending on the context and the nature. A memory leak on a seldom used service might not be as important as a vulnerability that would allow attackers to gain access to unauthorized information. Additionally, the frequency of a bug type can affect the decision of how important it is to go after. If a certain kind of crash, such as a null pointer error in Java, were happening hourly, then it might be more important to target than a bug of similar severity that occurs only once a year.

We have several means to collect data on the bugs that matter. First of all, the company maintains statistics on crashes and other errors that happen in production. Second, we have a "bug bounty" program, where people outside the company can report vulnerabilities on Facebook, or on apps of the Facebook family, e.g., Messenger, Instagram or WhatsApp. Third, we have an internal SEV initiative for tracking the most severe bugs that occur.

Our understanding of Bugs that Matter at Facebook drives our focus on advanced analyses. For contrast, a recent paper states: *"All of the static analyses deployed widely at Google are relatively simple, although some teams work on project-specific analysis frameworks for limited domains (such as Android apps) that do interprocedural analysis [17]"* and they give their entirely logical reasons, where we will explain why at Facebook we have made the decision to deploy interprocedural analysis (spanning multiple procedures) widely.

People and Deployments While not all bugs are the same, neither are all users, and we use different deployment models, depending on who intended audience (the people are that the analysis tool will be deployed to) is.

For classes of bugs that are intended for all or a wide variety of engineers on a given platform, we have gravitated towards a "diff time" deployment. In this deployment the analyzers participate as bots in code review, making automatic comments when an engineer submits a code modification. In Section 4 we recount a striking situation where the diff time deployment saw a 70% fix rate, where a more traditional "off line" or "batch" deployment (where bug lists are presented to engineers, outside their workflow) saw a 0% fix rate.

In case the intended audience is the much smaller collection of domain security experts in the company, we use two additional deployment models. At "diff time", security related issues are pushed to the security engineer on-call, so she can comment on an in-progress code change when necessary. Additionally, for finding all instances of a given bug in the codebase, or historical exploration, "off line" inspection provides a user interface for querying, filtering, and triaging all alarms.

In all cases, our deployments focus on the people that our tools serve and the way that they work.

Actioned Reports and Missed Bugs The goal of an industrial static analysis tool is to help people: at Facebook this is the engi-

neers, directly, and the people who use our products, indirectly. We have seen above how the deployment model can influence whether a tool is successful. Two concepts we use to understand this in more detail, and to help us improve our tools, are *actioned reports* and *observable missed bugs*.

The kind of action taken as a result of a reported bug depends on the deployment model as well as the type of bug. At diff time an action is an update to the diff that removes a static analysis report. In Zoncolan's off line deployment a report can trigger the security expert to create a task for the product engineer if the issue is important enough to follow up with the product team. Zoncolan catches more severe bugs than either manual security reviews or bug bounty reports. We measured that 43.3% of the severe security bugs are detected via Zoncolan. At the moment of writing, Zoncolan "action rate" is above 80% and we observed about 11 "missed bugs".

A "missed bug" is one that has been observed in some way, but that was not reported by an analysis. The means of observation can depend on the kind of bug. For security vulnerabilities we have bug bounty reports, security reviews or SEV reviews. For our mobile apps we log crashes and app not-responding events that occur on mobile devices.

The actioned reports and missed bugs are related to the classic concepts of true positives and false negatives from the academic static analysis literature. A true positive is a report of a potential bug that can happen in a run of the program in question (whether or not it will happen in practice); a false positive is one that cannot happen. Common wisdom in static analysis is that it is important to keep control of the false positives because they can negatively impact engineers who use the tools, as they tend to lead to apathy towards reported alarms. This has been emphasized, for instance, in previous CACM articles on industrial static analysis [1, 17]. False negatives, on the other hand, are potentially harmful bugs that may remain undetected for a long time. An undetected bug affecting security or privacy can lead to undetected exploits. In practice, fewer false positives often (though not always) implies more false negatives, and vice versa, fewer false negatives implies more false positives. For instance, one way to reign in false positives is to fail to report when you are less than sure that a bug will be real; but silencing an analysis in this way (say, by ignoring paths or by heuristic filtering) has the effect of missing bugs. And, if you want to discover and report more bugs you might also add more spurious behaviors.

The reason we are interested in advanced static analyses at Facebook might be understood in classic terms as saying: false negatives matter to us. However, it is important to note that the number of false negatives is notoriously difficult to quantify (how many unknown bugs are there?). Equally, though less recognized, the false positive rate is challenging to measure for a large, rapidly changing codebase: it would be extremely time consuming for humans to judge all reports as false or true as the code is changing.

Although true positives and false negatives are valuable concepts, we don't make claims about their rates and pay more attention to the action rate and the (observed) missed bugs.

Challenges: Speed, Scale and Accuracy A first challenge is presented by the sheer scale of Facebook's codebases, and the rate of change that they see. For the server-side we have over 100 million lines of Hack code, which Zoncolan can process in less than 30 minutes. Additionally, we have tens of millions of both mobile (Android and Objective C) code and backend C++ code. Infer processes the code modifications quickly (within 15 minutes on average) in its diff time deployment. All codebases see thousands of code modifications each day and our tools run on each code change. For Zoncolan, this can amount to analyzing 1 trillion lines of code per day.

- Advanced static analysis techniques performing deep reasoning about source code can scale to large industrial codebases, e.g. with 100 million lines.
- Static analyses should strike a balance between missed bugs (false negatives) and un-actioned reports (false positives).
- A "diff time" deployment, where issues are given to developers promptly as part of code review is important to catching bugs early and getting high fix rates.

Key Insights

It is relatively straightforward to scale program analyses that do simple checks on a procedure-local basis only. The simplest form are linters which give syntactic style advice (e.g. "the method you called is to be deprecated, please consider rewriting"). Such simple checks provide value and are in wide deployment in major companies including Facebook; we won't comment on them further in this article. But for more reasoning going beyond local checks, such as one would find in the academic literature on static analysis, scaling to 10s or 100s of millions of LOC is a challenge, as is the incremental scalability needed to support diff time reporting.

Infer and Zoncolan both deploy techniques similar to some of what one might find at the edge of the research literature. Infer (Section 4) deploys one analysis based on the theory of Separation Logic [16], with a novel theorem prover that implements an inference technique that guesses assumptions [5]. Another Infer analysis involves recently published research results on concurrency analysis [2, 10]. Zoncolan implements a new modular parallel taint analysis algorithm.

But how can Infer and Zoncolan scale? The core technical features they share are *compositionality* and *carefully crafted abstractions*. For most of the paper we will concentrate on what one gets from applying Infer and Zoncolan, rather than on their technical properties, but we outline their foundations in Section 6, and we include an appendix with additional material for the interested reader.

The challenge related to accuracy is intimately related to actioned reports and missed bugs. We try to strike a balance between these issues, informed by the desires based on the class of bugs and the intended audience. The more severe a potentially missed issue is, the lower the tolerance for missed bugs. Thus for issues that indicate a potential crash or performance regression in a mobile app such as Messenger, WhatsApp, Instagram, or Facebook, our tolerance for missed bugs is lower than for, e.g., stylistic lint suggestions (e.g., don't use deprecated method). For issues that could affect the security of our infrastructure or the privacy of the people using our products, our tolerance for false positives is higher still.

3. Software Development at Facebook

Facebook practices *continuous software development* [9], where a main codebase (master) is altered by thousands of programmers submitting code modifications (diffs). Master and diffs are the analogues of, respectively, GitHub master branch and pull requests. The developers share access to a codebase and they land, or commit, a diff to the codebase after passing code review. A *continuous integration system* (CI system) is used to ensure that code continues to build and passes certain tests. At this point, analyses run on the code modification and participate by commenting their findings directly in the code review tool.

The Facebook website was originally written in PHP, and then ported to Hack, a gradually typed version of PHP developed at Facebook (`https://hacklang.org/`). The Hack codebase spans over 100 million lines. It includes the web frontend, the internal web tools, the APIs to access the social graph from 1st and 3rd

party apps, the privacy-aware data abstractions, and the privacy-control logic for viewers and apps. Mobile apps – for Facebook, Messenger, Instagram and WhatsApp – are mostly written in Objective-C and Java. C++ is the main language of choice for backend services. There are tens of millions of lines each of mobile and backend code.

While they use the same development models, the website and mobile products are deployed differently. This affects: (i) what bugs are considered most important, and (ii) the way that bugs can be fixed. For the website, Facebook directly deploys new code to its own data centers, and bug fixes can be deployed directly to our data centers frequently, several times daily and immediately when necessary. For the mobile apps, Facebook relies on people to download new versions to from the Android or the Apple store; new versions are shipped weekly, but mobile bugs are less under our control because even if a fix is shipped it might not be downloaded to some people's phones.

Common runtime errors – *e.g.* null pointer exceptions, division by zero – are more difficult to get fixed on mobile than on the server. On the other hand, server-side security and privacy bugs can severely impact both the users of the Web version of Facebook as well as our mobile users, since the privacy checks are performed on the server-side. As a consequence, Facebook invests in tools to make the mobile apps more reliable and server-side code more secure.

4. Moving Fast with Infer

Infer is a static analysis tool applied to Java, Objective C and C++ code at Facebook [4]. It reports errors related to memory safety, to concurrency, to security (information flow), and many more specialized errors suggested by Facebook developers. Infer is run internally on the Android and iOS apps for Facebook, Instagram, Messenger and WhatsApp, as well as on our backend C++ and Java code.

Infer has its roots in academic research on program analysis with separation logic [5], research which led to a startup company (Monoidics Ltd) that was acquired by Facebook in 2013. Infer was open sourced in 2015 (www.fbinfer.com) and is used at Amazon, Spotify, Mozilla, and other companies.

Diff-time Continuous Reasoning Infer's main deployment model is based on fast incremental analysis of code changes. When a diff is submitted to code review an instance of Infer is run in Facebook's internal CI system (Figure 1). Infer does not need to process the entire code base in order to analyze a diff, and so is fast.

An aim has been for Infer to run in 15-20min on a diff on average, and this includes time to check out the source repository, to build the diff, and to run on base and (possibly) parent commits. It has typically done so, but we constantly monitor performance to detect regressions that makes it look longer, in which case we work to bring the running time back down. After running on a diff Infer then writes comments to the code review system. In the default mode used most often it reports only regressions: new issues introduced by a diff. The "new" issues are calculated using a bug equivalence notion which uses a hash involving the bug type and location-independent information about the error message, and which is sensitive to file moves and line number changes cause by refactoring, deleting or adding code; the aim is to avoid presenting warnings that developers might regard as pre-existing. Fast reporting is important to keep in tune with developers' workflows. In contrast, when Infer is run in whole-program mode it can take more than an hour (depending on the app), too slow for diff-time at Facebook.

Human Factors The significance of the diff-time reasoning of Infer is best understood by contrast with a failure. The first deployment was batch rather than continuous. In this mode Infer would be run once per night on the entire Facebook Android codebase, and it would generate a list of issues. We manually looked at the issues, and assigned them to the developers we thought best able to resolve them.

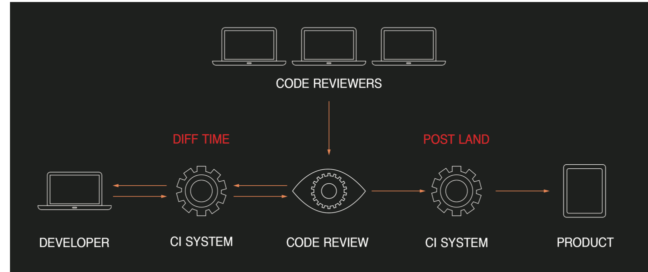

Figure 1: Continuous Development

The response was stunning: we were greeted by near silence. We assigned 20-30 issues to developers, and almost none of them were acted on. We had worked hard to get the *false positive rate* down to what we thought was less than 20%, and yet the *fix rate*—the proportion of reported issues that developers resolved—was near zero.

Next, we switched Infer on at diff time. The response of engineers was just as stunning: the fix rate rocketed to over 70%. The same program analysis, with same false positive rate, had much greater impact when deployed at diff time.

While this situation was surprising to the static analysis experts on the Infer team, it came as no surprise to Facebook's developers. Explanations that they offered to us may be summarized in the following terms.

One problem that diff-time deployment addresses is the *mental effort of context switch*. If a developer is working on one problem, and they are confronted with a report on a separate problem, then they must swap out the mental context of the first problem and swap in the second, and this can be time consuming and disruptive. By participating as a bot in code review, the context switch problem is largely solved: programmers come to the review tool to discuss their code with human reviewers, with mental context already swapped in. This illustrates as well how important timeliness is: if a bot were to run for an hour or more on a diff it could be too late to participate well.

A second problem that diff-time deployment addresses is *relevance*. When an issue is discovered in the codebase, it can be non-trivial to assign it to the right person. In the extreme, the issue might have been caused by somebody who has left the company. Furthermore, even if you think you have found someone familiar with the codebase, the issue might not be relevant to any of their past or current work. But, if we comment on a diff that introduces an issue then there is a pretty good (but not perfect) chance that it is relevant.

Mental context switch has been the subject of psychological studies [12], and it and the importance of relevance are part of the received collective wisdom impressed upon us by Facebook's engineers. Note that others have also remarked on the benefits of reporting during code review [17].

At Facebook we are working actively on moving other testing technologies to diff time when possible. We are also supporting academics on researching incremental fuzzing and symbolic execution techniques for diff time reporting.

Interprocedural Bugs Many of the bugs that Infer finds involve reasoning that spans multiple procedures or files. An example from OpenSSL illustrates.

```
1   @ThreadSafe
2   class RaceWithMainThread {
3     int mCount;
4     void protectedWriteOnMainThread_OK() {
5       OurThreadUtils.assertMainThread();
6       synchronized (this) { mCount = 1; }
7     }
8     int unprotectedReadOnMainThread_OK() {
9       OurThreadUtils.assertMainThread();
10      return mCount;
11    }
12    synchronized int protectedReadOffMainThread_OK() {
13      return mCount;
14    }
```

```
15    synchronized void
16    protectedWriteOffMainThread_BAD() {
17      mCount = 2;
18    }
19    int unprotectedReadOffMainThread_BAD() {
20      return mCount;
21    }
```

Figure 2: Top: simple example capturing a common safety pattern used in Android apps: threading information is used to limit the amount of synchronization required. As a comment from the original code explains: "`mCount` is written to only by the main thread with the lock held, read from the main thread with no lock held, or read from any other thread with the lock held." Bottom: unsafe additions to `RaceWithMainThread.java`.

```
apps/ca.c:2780: error: NULL_DEREFERENCE
pointer ‘revtm‘ last assigned on line 2778 could be null
and is dereferenced at line 2780, column 6

2778. revtm = X509_gmtime_adj(NULL, 0);
2779.
2780. i = revtm->length + 1;
```

The issue is that the procedure `X509_gmtime_adj()` can return null in some circumstances. Overall, the error trace found by Infer has 61 steps, and the source of null, the call to `X509_gmtime_adj` goes five procedures deep and it eventually encounters a return of null at call-depth 4. This bug was one of 15 that we reported to OpenSSL which were all fixed.

Infer finds this bug by performing *compositional reasoning*, which allows covering interprocedural bugs while still scaling to millions of lines of code. It deduces a precondition/postcondition specification approximating the behaviour of `X509_gmtime_adj`, and then uses that specification when reasoning about its calls. The specification includes 0 as one of the return values, and this triggers the error.

In 2017 we looked at bug fixes in several categories and found that for some (null dereferences, data races, and security issues) over 50% of the fixes were for bugs with traces that were interprocedural[1]. The interprocedural bugs would be *missed bugs* if we only deployed procedure-local analyses.

Concurrency A concurrency capability recently added to Infer, the RacerD analysis, provides an example of the benefit of feedback between program analysis researchers and product engineers [2, 15]. Development of the analysis started in early 2016, motivated by Concurrent Separation Logic [3]. After 10 months of work on the project, engineers from News Feed on Android caught wind of what we were doing and reached out. They were planning to convert part of Facebook's Android app from a sequential to a multi-threaded architecture. Hundreds of classes written for a single-threaded architecture had to be used now in a concurrent context: the transformation could introduce concurrency errors. They asked for interprocedural capabilities because Android UI is arranged in trees with one class per node. Races could happen via interprocedural call chains sometimes spanning several classes, and mutations almost never happened at the top level: procedural-local analysis would miss most races.

We had been planning to launch the proof tool we were working on in a year's time, but the Android engineers were starting their project and needed help sooner. So we pivoted to a *minimum viable product* which would serve the engineers – it had to be fast, with actionable reports, and not too many missed bugs on product code (but not on infrastructure code) [2, 15]. The tool borrowed ideas from Concurrent Separation Logic, but we gave up on the ideal of proving absolute race freedom. Instead, we established a 'completeness' theorem saying that, under certain assumptions, a theoretical variant of the analyzer reports only true positives [10]. The analysis checks for data races in Java programs; two concurrent memory accesses, one of which is a write. The example in Figure 2 (top) illustrates: If we run the Infer on this code it doesn't find a problem. The unprotected read and the protected write do not race because they are on the same thread. But, if we include additional methods that do conflict, then Infer will report races, as in Figure 2, bottom.

Impact. Since 2014 over 100,000 issues flagged by Infer have been resolved by Facebook's developers. The majority of Infer's impact comes from the diff-time deployment, but it is also run batch to track issues in master, issues addressed in fixathons and other periodic initiatives.

The RacerD data race detector saw over 2500 fixes in the year to March 2018. It supported the conversion of Facebook's Android app from a single-threaded to a multi-threaded architecture by searching for potential data races, without the programmers needing to insert annotations for saying which pieces of memory are guarded by what locks. This conversion led to an improvement in scroll performance and, speaking about the role of the analyzer, Benjamin Jaeger, an Android engineers at Facebook, stated[2]: *"without Infer, multithreading in News Feed would not have been tenable"*. As of March 2018, *no Android data race bugs missed by Infer had been observed in the previous year* (modulo 3 analyzer implementation errors [2]).

The fix rate for the concurrency analysis to March 2018 was roughly 50%, lower than for the general diff analysis previously. Our developers have emphasized that they appreciate the reports because concurrency errors are difficult to debug. This illustrates our earlier points about balancing action rates and bug severity. See [2] for more discussion on fix rates.

Overall, Infer reports on over 30 types of issue, ranging from deep inter-procedural checks to simple procedure-local checks and lint rules. Concurrency support includes checks for deadlocks and starvation, with hundreds of "app not-responding"' bugs being fixed in the past year. Infer has also recently implemented a security analysis (a 'taint' analysis), which has been applied to Java and

[1] https://code.facebook.com/posts/1537144479682247/finding-interprocedural-bugs-at-scale-with-infer-static-analyzer/

[2] https://code.facebook.com/posts/1985913448333055/multithreaded-rendering-on-android-with-litho-and-infer/

C++ code; it gained this facility by borrowing ideas from Zoncolan, which we describe next.

5. Staying Secure with Zoncolan

One of the original reasons for the development and adoption of Hack was to enable more powerful analysis of the core Facebook codebase. Zoncolan is the static analysis tool we built to find code and data paths that may cause a security or a privacy violation in our Hack codebase.

The code in Fig. 3 is an example of a vulnerability prevented by Zoncolan. If the `member_id` variable on line 21 contains the value `../../users/delete_user/` it is possible to redirect this form into any other form on Facebook. On submission of the form, it will invoke a request to `https://facebook.com/groups/add_member/../../users/delete_user/`, which will delete the user's account. The root cause of the vulnerability in Fig. 3 is that the attacker controls the value of the `member_id` variable which is used in the `action` field of the `<form>` element. Zoncolan follows the interprocedural flow of untrusted data (*e.g.*, user-input) to sensitive parts of the codebase. Virtual calls do make interprocedural analysis hard since the tool in general does not know the precise type of an object. To avoid missing paths (and thus bugs), Zoncolan must consider all the possible functions a call may resolve to.

SEV-Oriented Static Analysis Development We designed and developed Zoncolan in collaboration with the Facebook App Security team. Alarms reported by Zoncolan are inspired by security bugs that the App Security team uncovered.

The initial design of Zoncolan began with a list of severe bugs (*SEV* in Facebook terminology) that were provided to us by security engineers. For each bug we asked ourselves: "*How could we have caught it with static analysis?*". Most of those historical bugs were no longer relevant because the programming language or a secure framework prevented them from recurring—for instance, the widespread adoption of XHP made it possible to build XSS-free web pages by construction. We realized that the remaining bugs involved interprocedural flows of untrusted data, either directly or indirectly, into some privileged APIs. Detecting such bugs can be automated with static taint flow analysis [18] which tracks how the data originating from some untrusted sources reaches *or influences* the data reaching some sensitive parts of the codebase (sinks).

When a security engineer discovers a new vulnerability we evaluate whether that class of vulnerability is amenable to static analysis. If it is, we prototype the new rule, iterating with the feedback of the engineer in order to refine results to strike the right balance of false postives/false negatives. When we together believe the rule is good enough, it is enabled on all runs of Zoncolan in production. We adopt the standard Facebook App Security severity framework, which associates to each vulnerability an impact level, in a scale from 1 (best-practice) to 5 (SEV-worthy). A security impact level of 3 or more is considered as *severe*.

Scaling the analysis A main challenge was to scale Zoncolan to a codebase of more than 100 millions of lines of code. Thanks to a new parallel, compositional, non-uniform static analysis that we designed, Zoncolan performs the full analysis of the code base in less than 30 minutes, on a 24 core server.

Zoncolan builds a dependency graph that relates methods to their potential callers. It uses this graph to schedule parallel analyses of individual methods. In the case of mutually recursive methods, the scheduler iterates the analysis of the methods until it stabilizes, *i.e.*, no more flows are discovered. Suitable operators (called widenings in the static analysis literature, [7]) ensure the convergence of the iterations. It is worth mentioning that, even though the concept of taint analysis is well established in Academia, we had to develop new algorithms in order to scale to the size of our codebase.

Funneled Deployment Figure 4 provides a graphical representation of the Zoncolan deployment model. This funneled deployment model optimizes bug detection with the goal of supporting security of Facebook: The Zoncolan master analysis finds all existing instances of a newly discovered vulnerability. The Zoncolan diff analysis avoids vulnerabilities from being (re-)introduced in the codebase.

Zoncolan periodically analyzes the entire Facebook Hack codebase to update the master list. The target audience is security engineers performing security reviews. In the master analysis, we expose *all* alarms found. Security engineers are interested in all existing alarms for a given project or a given category. They triage alarms via a dashboard which enables filtering by project, code location, source and/or destination of the data, length or features of the trace. When a security engineer finds a bug, he/she files a task for the product group and provides guidance on how to make the code secure. When an alarm is a false positive, he/she files a task for the developers of Zoncolan with an explanation of why the alarm is false. The Zoncolan developers then refine the tool to improve the precision of the analysis. After a category has been extensively tested, the Zoncolan team, in conjunction with the App security team, evaluates if it can be promoted for diff analysis. Often promotion involves improving the signal by filtering the output according to *e.g.*, the length of the inter-procedutal trace, the visibility of the endpoint (external or internal?), etc. At the moment of writing circa $1/3$ of the Zoncolan categories are enabled for diff analysis.

Zoncolan analyzes every Hack code modification and reports alarms if a diff introduces new security vulnerabilities. The target audience is: (i) the author and the reviewers of the diff (Facebook software engineers who are not security experts), and (ii) the security engineer in the on-call rotation (who has a limited time budget). When appropriate, the on-call validates the alarm reported, blocks the diff, and provides support to write the code in a secure way. For categories with very high signal, Zoncolan acts as a security bot: it by-passes the security on-call and instead comments directly on the diff. It provides a detailed explanation on the security vulnerability, how it can be exploited, and it includes references to past incidents, *e.g.* SEVs.

Finally, note that the funneled deployment model makes it possible to scale up the security fixes, without reducing the overall coverage Zoncolan achieves (i.e., without missing bugs): If Zoncolan determines a new issue is not high-signal enough for auto-commenting on the diff, but needs to be looked at by an expert, it pushes it to the on-call queue. If the alarm makes neither of these cuts, the issue will end up in the Zoncolan master analysis after the diff is committed.

Impact Zoncolan has been deployed for more than two years at Facebook, first to security engineers, then to software engineers. It has prevented thousands of vulnerabilities from being introduced to Facebook's codebase. Figure 5 compares the number of severe bugs, *i.e.* bugs of severity 3 to 5, prevented by Zoncolan, in a 6 months period, to the traditional programs adopted by security engineers, such as manual code reviews/pentesting and bug bounty reports. The bars show that *at Facebook, Zoncolan catches more severe bugs than either manual security reviews or bug bounty reports. We measured that* 43.3% *of the severe security bugs are detected via Zoncolan.*

The graph in Fig. 6 shows the distribution of the *actioned* bugs found by Zoncolan at different stages of the deployment funnel, according to the security impact level. The largest number of categories are enabled for the master analysis, so it is not unexpected that it is the largest bucket. However, when restricting to severe

```
1  <?hh
2  class AddMemberToGroup extends FacebookEndpoint {
3    private function getIDs (): (string, int) {
4      // User input, untrusted
5      return tuple((string) $this->getRequest('member_id'),
6                   (int) $this->getRequest('gid'));
7    }
8
9    public function render(): :xhp {
10     list($member_id, $group_id) = $this->getIDs();
11     return this->getConfirmationForm($group_id, $member_id);
12   }
13
14   public function getConfirmationForm
15    (int $group_id, string $member_id): :xhp {
16     $url = "https://facebook.com/groups/add_member/" .
17            $member_id;
18
19     return
20       <form method="post" action={$url}>
21         <input name="gid" value={$group_id}/>
22         <input name="action" value="add"/>
23       </form>;
24   }
25 }
```

Figure 3: Example of a bug that Zoncolan prevents. It may cause the attacker to delete a user account. The attacker can provide an input on line 5 that causes a redirection to any other form on Facebook at line 20.

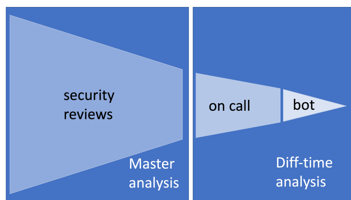

Figure 4: Funneled Deployment of Zoncolan

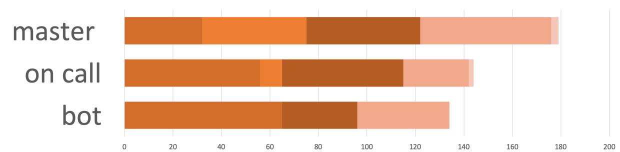

Figure 6: Distribution of *all* the bugs fixed, in a 6 months period, based on Zoncolan's Funneled Deployment and bug severity (darker implies more severe).

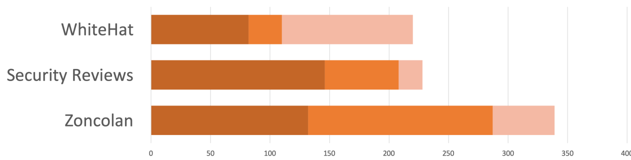

Figure 5: Comparison of *severe* bugs reported by Zoncolan w.r.t. Security reviews and Bug Bounty, in a 6 months period (darker implies more severe).

bugs the diff analysis largely overtakes the master analysis: 211 severe issues are prevented at diff-time, versus 122 detected on master. Overall, we measured the ratio of Zoncolan actioned bugs to be close to 80%.

We also use the traditional security programs to measure missed bugs, *i.e.*, the vulnerabilities for which there is a Zoncolan category, but the tool failed to report them. To date, *we have had about* 11 *missed bugs*, some of them caused by a bug in the tool or incomplete modelling.

6. Compositionality and Abstraction

The technical features that underpin our analyses are *compositionality* and *abstraction*.

The notion of compositionality comes from language semantics: A semantics is compositional if the meaning of a compound phrase is defined in terms of the meanings of its parts and a means of combining them. The same idea can be applied to program analysis [8, 5]. A program analysis is compositional if the analysis result of a composite program is defined in terms of the analysis results of its parts and a means of combining them. When applying compositionality in program analysis, there are two key questions:

a) How to represent the meaning of a procedure concisely

b) How to combine the meanings in an effective way

For a) we need to approximate the meaning of a component by abstracting away the full behavior of the procedure and to focus only on the properties relevant for the analysis. For instance, for security analysis, one may be only interested that a function returns a user-controlled value, when the input argument contains a user-controlled string, discarding the effective value of the string. More formally, the designer of the static analysis defines an appropriate mathematical structure, called the *abstract domain* [7], which allows to approximate this large function space much more succinctly. The design of a static analysis relies on abstract domains which are precise enough to capture the properties of interest and coarse enough to make the problem computationally tractable. The 'abstraction of a procedure meaning' is often called a *procedure summary* in the analysis literature [19].

The answer to question b) mostly depends on the specific abstract domain chosen for the representation of summaries. Further information on the abstractions supported by Infer and Zoncolan, as well as brief information on recursion, fixpoints, and analysis

algorithms, may be found in the technical appendix. It is worth discussing the intuitive reason for why compositional analysis *together with* crafted abstract domains can scale: each procedure only needs to be visited a few times, and furthermore many of the procedures in a codebase can be analyzed independently, thus opening up opportunities for parallelism. A compositional analysis can even have a runtime that is (modulo mutual recursion) a linear combination of the times to analyze the individual procedures. For this to be effective, the cost of analyzing a single procedure should also be contained by a suitable abstract domain, for instance limiting or avoiding disjunctions.

Finally, compositional analyses are naturally incremental: changing one procedure does not necessitate re-analyzing all other procedures. This is important for fast diff-time analysis.

7. Conclusion

In this paper we have described how, as static analysis people working inside Facebook, we have developed program analyses in response to the needs that arise from production code and engineers' requests. This presents different constraints and opportunities than situations where an analysis is developed to be marketed to other companies. Facebook has enough important code and problems that it is worthwhile to have embedded teams of analysis experts, and we have seen (e.g., in the use of Infer to support multi-threaded Android News Feed, and in the evolution of Zoncolan to detect SEV-worthy issues) how this can lead to impact for the company. Although our primary responsibilty is to serve the company, we believe that our learnings and techniques can generalize beyond the specific industrial context. E.g., Infer is used at other companies such as Amazon, Mozilla and Spotify, we have produced new scientific results [2, 10], and proposed new scientific problems [14, 11]. Indeed, our impression as (former) researchers working in an engineering org is that having science and engineering playing off one another in a tight feedback loop is possible, even advantageous, when practicing static analysis in industry at present.

In closing, to people in industry we say: advanced static analyses, like those found in the research literature, can be deployed at scale and deliver value for general code. And to academics we say: from an industrial point of view the subject appears to have many unexplored avenues, and this provides opportunities for research to inform future tools.

Acknowledgements

Special thanks to Ibrahim Mohamed for being a tireless advocate for Zoncolan among security engineers, to Cristiano Calcagno for leading Infer's technical development for several years, and to our many teammates and other collaborators art Facebook for their contributions to our collective work on scaling static analysis.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, A. Hallem, C.-H. Gros, A. Kamsky, A. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM*, 53(2):66–75, 2010.

[2] S. Blackshear, N. Gorogiannis, I. Sergey, and P. O'Hearn. Racerd: Compositional static race detection. In *OOPSLA*, 2018.

[3] S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P.W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11, 2015.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[6] B. Cook. Formal reasoning about the security of amazon web services. In *LICS*, pages 38–47, 2018.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th POPL, pp238-252, 1977.

[8] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

[9] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *Internet Computing, IEEE*, 17(4):8–17, 2013.

[10] N. Gorogiannis, I. Sergey, and P. O'Hearn. A true positives theorem for a static race detector. In *POPL*, 2019.

[11] M. Harman and P. O'Hearn. From start-ups to scale-ups: Open problems and challenges in static and dynamic program analysis for testing and verification). In *SCAM*, 2018.

[12] S. T. Iqbal and E. Horvitz. Disruption and recovery of computing tasks: field study, analysis, and directions. In *CHI*, pages 677–686, 2007.

[13] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. D. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[14] P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *LICS*, 2018.

[15] P. W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In *SAS*, pages 56–70, 2018.

[16] P. W. O'Hearn. Separation logic. *Comm. ACM*, 62(2):86–95, 2019.

[17] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *CACM*, 2018.

[18] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[19] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, 2008.

A. Appendix

In the main body of the paper we explained how our anlayzers scale because of their foundation based on compositionality and crafted abstract domains. This technical appendix provides some information on our crafted abstractions, and on the analysis algorithms we use.

A.1. Infer

Infer.Classic The original version of Infer, which we dub Infer.Classic in this section, sprung out of academic research on program analysis with separation logic [5], where the program analyzer manipulates logical assertions called *symbolic heaps* describing computer memory.

Infer.Classic finds memory safety issues spanning multiple procedures/files, as it attempts to construct program proofs that stitch the results together for the constituent procedures. The summaries are precondition/postcondition specifications, where the preconditions attempt to describe the *footprint* of a procedure: the memory cells it accesses [16]. Note that there will in general be a great many pre/post specs that are true for a procedure; considering footprints instead of arbitrary preconditions greatly cuts down the possibilities.

The focus on the footprint, rather than the global state of the program, provides an approach to question a) in Section 6, which concerns concise representations of procedure meanings. Part b), the stitching together of results, is implemented using a novel logical principle, *bi-abduction*; see [5].

Let's consider the C code in Figure 7. `malloc_int_wrapper`, by mimicking the behaviour of `malloc`, gets as summary a disjunction in the post-condition:

$$\{emp\}\, \texttt{malloc_int_wrapper()}\, \{return == 0\} \vee \{return \mapsto -\}.$$

By knowing that an assignment to `*x` requires the memory pointed-to by `x` to be allocated, Infer computes the following summary for `set`:

$$\{x \mapsto -\}\, \texttt{set(int *x, int n)}\ \ \{x \mapsto n\}$$

The summary says that to run the procedure without crashing, x must point to an allocated cell (i.e., $x \mapsto -$) and, at the end, the value of that cell will be n.

Infer uses and composes these summaries for the analysis of the procedure `caller`. After line 6 the analysis obtains two assertions, resulting form the disjuncitons in the post of `malloc_int_wrapper`:

$$y \mapsto - \qquad \text{and} \qquad y == 0$$

If we unify `y` with the `x` parameter of `set(int *x, int n)`, then the first of these assertions is equivalent to the precondition of its summary, $x \mapsto -$. However, the second assertion after unification becomes $x == 0$ which is inconsistent with the required precondition $x \mapsto -$. We cannot safely call the `set` procedure from the symbolic heap `y == 0`, and cannot complete the proof of the overall procedure `caller`. Infer uses this failed proof information to issue an error message[3].

If we uncomment line 7 then no error is reported and the proof goes through.

The use of `malloc_int_wrapper` in this example is instructive: Accurate memory analysis at scale tends to need disjunctions for precise-enough summaries. On one hand, it is not uncommon to find `malloc` wrappers in the wild; for instance, we reported several issues to openssl which involved a much more complex `malloc` wrapper, `OPENSSL_MALLOC`[4], and both the discovered summaries need to be expressive enough to handle such examples. On the other hand, even when one does not literally have a malloc wrapper it is not uncommon for a procedure to return valid allocated pointers in some circumstances and 0 in others.

```
1  void set(int *x, int n) { *x =n; }
2
3  int* malloc_int_wrapper() { return malloc(sizeof(int)); }
4
5  int caller () {
6      int* y = malloc_int_wrapper();
7      // if (!y) {return 42;};
8      set(y, 3);
9      free(y);
10 }

Infer error message:
pointer y last assigned on line 6 could be null and is
dereferenced by call to set() at line 8, column 5
```

Figure 7: Infer.Classic example

RacerD: an Infer.AI While Infer.Classic produced impact, perhaps its greater contribution was to establish that advanced reasoning techniques would survive and even thrive in Facebook's high momentum software development environment. This emboldened us to move forward with other techniques. Sam Blackshear created an analysis framework, Infer.AI, which facilitates building *compositional abstract interpreters* as in Section 6. This section discusses a specific Infer.AI, RacerD.

RacerD detects data races in Java programs; two concurrent memory accesses, one of which is a write. The example in Figure 2 (top) illustrates some of RacerD's ideas. If we run the analyzer on this code it doesn't find a problem. The unprotected read and the protected write do not race with one another because they are known to be on the same thread. This is one way that developers try to write thread safe code while avoiding synchronization for performance reasons.

Technically, RacerD works by first computing a summary for each method in a class, which records potential accesses together with information such as whether they are protected by a lock or confined to a thread. Next, RacerD looks through all the summaries for the class in question checking for potential conflicts. In this example, the summaries record an approximation of the information that the accesses to `mCount` are both protected by being on the same thread:

```
protectedWriteOnMainThread_OK()
Thread: true, Lock: true, Write to this.RaceWithMainThread.mCount:6
unprotectedReadOnMainThread_OK()
Thread: true, Lock: false, Read of this.RaceWithMainThread.mCount:10
protectedReadOffMainThread_OK()
Thread: false, Lock: true,  Read of this.RaceWithMainThread.mCount:13
```

RacerD concludes that a race is not possible because the threading information indicate mutual exclusion.

On the other hand, if we include additional methods that do conflict, then RacerD will report potential races, as in Figure 2, bottom. The summaries for the additional methods record information about the accesses as follows:

```
protectedWriteOffMainThread_BAD()
Thread: false, Lock: true, Write to this.RaceWithMainThread.mCount:17
unprotectedReadOffMainThread_BAD()
Thread: false, Lock: false, Read of this.RaceWithMainThread.mCount:20
```

[3] In fact, Infer records somewhat more than the bare pre/post information in summaries, to be able to report the line of the dereference in `set` when showing the user an error trace

[4] `rt.openssl.org/Ticket/Display.html?id=3403&user=guest&pass=guest`

The summary for `protectedWriteOffMainThread_BAD` shows the potential conflict with the unprotected read on the main thread. One is protected by a lock, and the other is protected by knowledge that it is on the main thread, but these are not sufficient to provide mutual exclusion: RacerD reports a race between this access and the one at line 8. Similarly, the access at line 20 in `unprotectedReadOffMainThread_BAD` is protected by neither a lock nor a thread. RacerD reports races with both the access at line 6 and the access at line 17.

RacerD employs a crafted abstraction oriented to finding races rather than memory safety (as with Infer.Classic). (i) it uses sets of accesses rather than pre/post specs as the summaries; (ii) it does not maintain any disjunctions, taking the union of accesses in branches of if statements. A consequence of using this crafted abstraction is that RacerD is blazingly fast: e.g., it can analyze 10k LOC in under 2 seconds [2].

The choice to avoid disjunctions entails a precision loss, and leads to false positives. Programs that race or not depending on boolean conditions can trip up the analysis. A typical example is the implementation of ownership transfer, where (say) you set a value to indicate that you are going to access an object outside of synchronization, code in other synchronization blocks then avoids to access the object: RacerD can easily report false positive races in such cases. In Facebook's Android code, fine-grained idioms like this are present in infrastructure code, but much less so in product code. For instance, Facebook's Litho UI library has fine-grained examples that lead to false positives of this variety in Infer, but the Litho authors advised us not to concentrate on the fine-grained idioms, to in a sense go against our first instincts as analysis experts to pursue subtle examples: they advised to do a better job with the coarser uses of concurrency typical in our product code (the majority), for which the precision loss was found to be acceptable. See [2, 15] for further discussion.

A.2. Zoncolan

Zoncolan performs a full program analysis of 100 million lines of Hack code in less than 30 minutes. To scale up to such a code base, Zoncolan uses a parallel compositional analysis in conjunction with a non-uniform abstract domain to approximate the flow of dangerous information. The code in Fig. 3 is an example of a vulnerability prevented by Zoncolan. Zoncolan needs to follow the interprocedural flow of untrusted data (*e.g.*, user-input) to sensitive parts of the code base, approximating virtual calls. Zoncolan utilizes a dependency graph to resolve the virtual calls and to schedule the analysis of individual functions, that will be analyzed in parallel.

For each function, Zoncolan performs a *forward* analysis, to propagate tainted data to the exit point of the method, and a *backward* analysis to propagate sinks to the entry arguments.

The forward analysis of `getIDs` (Fig. 8) infers that the user input at lines 5 and 6 flows to the first and second component of the pair returned by that method.

The backward analysis of `getConfirmationForm` (Fig. 9) states that a tainted value for the argument `$member_id` will reach the `action` field of `<form>` at line 20 after a string concatenation, and the argument `$group_id` since values reaching the `input::value` field do not pose any security threat.

The summary also illustrates the non-uniform abstraction being used: The exact literal string being concatenated with `$member_id` is abstracted to a single bit "via stringconcat" and unlike what we do for sources and sinks, Zoncolan does not retain the program location where the concatenation happens (*i.e.*, line 16). Finally, in the function `render` Zoncolan utilizes compositional reasoning to stich together the pieces of the flow from the input to the form action, obtaining the trace in Fig. 10. The summaries computed by Zoncolan contain just enough call-edge information to be able to reproduce full traces when displaying the alarms.

A.3. Recursion, Fixpoints, etc.

The early versions of Infer, which was derived from [5], worked by constructing a call graph, which which was used to schedule a bottom-up analysis algorithm where called are analyzed before callers. Cycles in the graph were broken arbitrarily to find a starting point. Summaries were stored in cache, which was consulted when analyzing a code modification in incremental fashion. This version of Infer implemented a shape analysis, one of the more expensive forms of analysis even intra-procedurally, and timeouts were used to ensure that the local analyses terminated (in that case, delivering a $\top$ result in the jargon).

In 2015, Cristiano Calcagno replaced the bottom-up Infer backend with one that works "on demand" instead of bottom up, and which does not require prior computation of a call graph. The analysis has a "begin anywhere" property, where can start anywhere in the codebase, irrespective of caching. In case the analysis needs a procedure summary and it is not in cache, the analyzer is called recursively to produce the summary. The on demand mode allowed for additional parallelism, and led to non-trivial performance gains over the bottom-up implementation. Infer.Classic and Infer.AI both use on-demand, presently.

Infer has used a bounded approach to (possibly mutual) recursion. We have experimented with different amounts of recursive unwinding, and at present Infer stops after one iteration. This choice is consistent with Infer's use to prevent regressions (like a testing tool), but not full proof. (For cognoscenti, mathematically this is like saying to calculate $F^i(\bot)$ for a given i and take that as the summary whether or not it is a fixpoint of F, with a slight modification for mutual recursion. This can be seen as a version of bounded symbolic model checking, which calculates an over-approximation of a finite unwinding of a program.)

As we indicated in the main body of the paper, in the case of mutually recursive functions Zoncolan iterates the analysis until the function summaries reach a fixpoint. To enforce convergence, Zoncolan uses a widening operator [7].

Infer and Zoncolan build on basic ideas from the research literature on compositional program analysis [8, 5], but there appears to be much valuable work to be done in both the theory and practice of algorithms in this area, especially when it comes to the scaling properties of compositional algorithms.

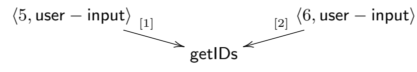

Figure 8: Summary for `GenericGroupForm::getIDs`

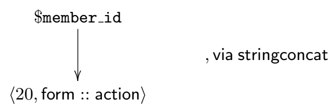

Figure 9: Summary for `getConfirmationForm`

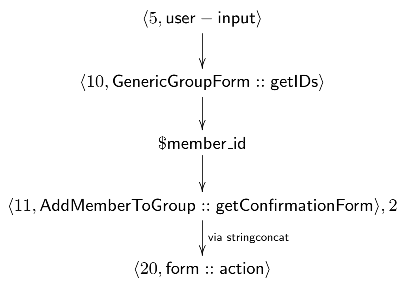

Figure 10: Trace to the vulnerability