Moving Fast with Software Verification

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez

Facebook Inc.

Abstract. For organisations like Facebook, high quality software is important. However, the pace of change and increasing complexity of modern code makes it difficult to produce error-free software. Available tools are often lacking in helping programmers develop more reliable and secure applications.

Formal verification is a technique able to detect software errors statically, before a product is actually shipped. Although this aspect makes this technology very appealing in principle, in practice there have been many difficulties that have hindered the application of software verification in industrial environments. In particular, in an organisation like Facebook where the release cycle is fast compared to more traditional industries, the deployment of formal techniques is highly challenging.

This paper describes our experience in integrating a verification tool based on static analysis into the software development cycle at Facebook.

1 Introduction

This is a story of transporting ideas from recent theoretical research in reasoning about programs into the fast-moving engineering culture of Facebook. The context is that most of the authors landed at Facebook in September of 2013, when we brought the INFER static analyser with us from the verification startup Monoidics [4, 6]. INFER itself is based on recent academic research in program analysis [5], which applied a relatively recent development in logics of programs, separation logic [10]. As of this writing INFER is deployed and running continuously to verify select properties of every code modification in Facebook's mobile apps; these include the main Facebook apps for Android and iOS, Facebook Messenger, Instagram, and other apps which are used by over a billion people in total.

In the process of trying to deploy the static analyser the most important issue we faced was integration with Facebook's software development process. The software process at Facebook, and an increasing number of Internet companies, is based on fast iteration, where features are proposed and implemented and changed based on feedback from users, rather than wholly designed at the outset. The perpetual, fast, iterative development employed at Facebook might seem to be the worst possible case for formal verification technology, proponents of which sometimes even used to argue that programs should be developed only after a prior specifications had been written down. But we found that verification technology can be effective if deployed in a fashion which takes into account when and why programmers expect feedback. INFER runs on every "diff", which is a code change submitted by a developer for code review. Each day a number of bugs are reported on diffs and fixed by developers, before the diff is committed and eventually deployed to phones. Technically, the important point is that INFER is a compositional[1]program analysis, which allows feedback to be given to developers in tune with their flow of incremental development.

2 Facebook's Software Development Model

Perpetual development. As many internet companies Facebook adopts a continuous development model [9]. In this model, software will never be considered a *finished product.* Instead features are continuously added and adapted and shipped to users. Fast iteration is considered to support rapid innovation. For its web version, Facebook pushes new changes in the code twice a day.

This perpetual development model fits well with the product and its use-case. It would be impossible to foresee a-priori how a new feature would be used by the hundreds of million of people using Facebook services every day. The different uses influence the way a new feature is shaped and further developed. In other words, Facebook prioritises people using the product rather than an initial design proposed in some fixed specification by architects at the company.

Perpetual development on mobile versus web. In the last couple of years, Facebook has gone through a shift. From being a web-based company Facebook transitioned to embrace mobile. Use of its mobile applications on the Android and iOS platforms has increased substantially, reflecting a global trend for consumers of Internet content.

For mobile applications, Facebook applies a continuous development model as well. However there are some fundamental differences w.r.t. web development. Although the development cycle is the same, the deployment is fundamentally different. In web development the software runs on Facebook servers in our datacenters, and in the client on code downloaded from our servers by a browser. New code can, therefore, be deployed directly to the servers, which then serves the users (including by serving them Javascript); new versions of the software are deployed without the users getting involved.

On the contrary, mobile applications run on users' phones. Therefore, it is up to the user to update to a new version of the app implementing new features or fixing existing bugs. Facebook can only distribute a new version to the Apple

[1] A compositional analysis is one in which the analysis result of a composite program is computed from the results of its parts. As a consequence, compositional analyses can run on incomplete programs (they are not whole-program analyses), are by their nature incremental, scale well, and tolerate imprecision on parts of code that are difficult to analyse [5].

App Store or Google Play, but Facebook is not anymore in control of which version a user is running on her mobile device.

This difference has dramatic impact on bug fixes. On web when a bug is discovered a fix can be shipped to the servers as part of a periodic release or in exceptional cases immediately via a "hotfix". And on the web mechanisms exist to automatically update the JavaScript client software running in the browser, allowing fixes to quickly and automatically be deployed as soon as they have been developed. On current mobile platforms updates must typically be explicitly authorised by the device owner, so there is no guarantee that a fix will be deployed in a timely manner, if ever, once it is developed.

The sandboxes provided by modern web browsers also make it easier to isolate the effects of a bug in one part of the interface from another, allowing the experience to gracefully degrade in the face of runtime errors. Current mobile platforms typically provide a model closer to processes running on a traditional operating system and will often terminate the entire app when a runtime error is detected in any part of it. This lower fault tolerance increases the potential severity of bugs which would have a minor impact on the web.

Thus mobile development at Facebook presents a strong dichotomy: on one hand it employs continuous development; on the other hand it could benefit from techniques like formal verification to prevent bugs before apps are shipped.

When the INFER team landed at Facebook there was a well developed version of INFER for C programs, and a rudimentary version for Java. Facebook has considerable amounts of C++, Javascript, php, objective-C and Java code, but less development is being done in pure C. This, together with the above discussion determined our first targets, Android and iPhone apps.

3 Software Verification in the Perpetual Development Era

As we have seen, Facebook employs a perpetual development model both for web and mobile software. While in such an environment it is difficult to envisage requiring specs to always be written before programming starts, a common approach in static analysis has been to work towards the implicit specification that (certain) runtime errors cannot occur. Of course, when an assertion is placed into code it can help the analysis along. In INFER's case at the beginning the implicit safety properties were null pointer exceptions and resource leaks for Android code, and additionally memory leaks for iOS.

Unlike many other software companies, Facebook does not have a separate quality assurance (QA) team or professional testers. Instead, engineers write (unit) tests for their newly developed code. But as a part of the commit and push process there is a set of regression tests that are automatically run and the code must pass them before it can be pushed. This juncture, when diffs are reviewed by humans and by tests, is a key point where formal verification techniques based on static analysis can have impact.

There are several features that the verification technique should offer to be adopted in such different environment:

- *Full automation and integration.* The technique should be push-button and integrated into the development environment used by programmers.
- *Scalability.* The technique scales to millions of lines of code.
- *Precision.* Developers' time is an important resource. An imprecise tool providing poor results would be seen as a waste of that resource.
- *Fast Reporting.* The analysis should not get in the way of the development cycle; therefore it has to report to developers in minutes, before programmers commit or make further changes. As we will see in Section 5, fast reporting is not only about analysing code fast, but it also involves good integration with the existing infrastructure where many other tasks need to be performed.

These requirements are challenging. In our context we are talking about analysis of large Android and iPhone apps (millions of lines of code are involved in the codebases). The analysis must be able to run on thousands of code diffs in a day, and it should report in under 10 minutes on average to fit in with the developer workflow. There are intra-procedural analyses and linters which fit these scaling requirements, and which are routinely deployed at Facebook and other companies with similar scale codebases and workflows. But if an analysis is to detect or exclude bugs involving chains of procedure calls, as one minimally expects of verification techniques, then an inter-procedural analysis is needed, and making inter-procedural analyses scale to this degree while maintaining any degree of accuracy has long been a challenge.

4 Background: the INFER Static Analyser

INFER [4] is a program analyser aimed at verifying memory safety and developed initially by Monoidics Ltd. It was first aimed at C code and later extended to Java. After the acquisition of Monoidics by Facebook, INFER's development now continues inside Facebook.

INFER combines several recent advances in automatic verification. It's underlying formalism is separation logic [10]. It implements a compositional, bottom-up variant of the classic RHS inter-procedural analysis algorithm based on procedure summaries [11]. There are two main novelties. First, it uses compact summaries, based on the ideas of footprints and frame inference [2] from separation logic, to avoid the need for huge summaries that explicitly tabulate most of the input-output possibilities. Second, it uses a variation on the notion of abductive inference to discover those summaries [5].

Bi-abduction. INFER computes a compositional shape analysis by synthesising specification for a piece of code in isolation. Specifications in this case are Hoare's triples where pre/post-conditions are separation logic formulae. More specifically, for a given piece of code C, INFER synthesises pre/post specifications of the form

$$\{P\}\ C\ \{Q\}$$

by inferring suitable P and Q. A crucial point is that such specifications do not express functional correctness but rather memory safety. The consequence is that they relates to a basic general property that every code should satisfy.

The theoretical notion allowing INFER to synthesise pre and post-conditions in specifications is *bi-abductive inference* [5]. Formally, it consists in solving the following extension of the entailment problem:

$$H * A \vdash H' * F$$

where H, H' are given formulae in separation logic describing a heap configuration whereas F (frame) and A (anti-frame) are unknown and need to be inferred. Bi-abductive inference is applied during an attempted proof of a program to discover a collection of anti-frames describing the memory needed to execute a program fragment safely (its footprint).

Triples of procedures in a program are composed together in a bottom-up fashion according to the call graph to obtain triples of larger pieces of code.

Soundness. The soundness property for the algorithm underlying INFER is that if INFER finds a Hoare triple $\{P\}\ C\ \{Q\}$ for a program component C then that triple is true in a particular mathematical model according to the fault-avoiding interpretation of triples used in separation logic [10]: any execution starting from a state satisfying P will not cause a prescribed collection of runtime errors (in the current implementation these are leaks and null dereferences) and, if execution terminates, Q will be true of the final state. Soundness can also be stated using the terminology of abstract interpretation (see [5], section 4.4).

Soundness can never be absolute, but is always stated with respect to the idealization (assumptions) represented by a mathematical model. In INFER's case limitations to the model ([5]) include that it doesn't account for the concurrency or dynamic dispatch found in Android or iPhone apps. So interpreting the results in the real world must be done with care; e.g., when an execution admits a race condition, INFER's results might not over-approximate. Note that these caveats are given even prior to the question of whether INFER correctly implements the abstract algorithm. Thus, soundness does not translate to "no bugs are missed." The role of soundness w.r.t. the mathematical model is to serve as an aid to pinpoint what an analysis is doing and to understand where its limitations are; in addition to providing guarantees for executions under which the model's assumptions are met.

Context In this short paper we do not give a comprehensive discussion of related work, but for context briefly compare INFER to several other prominent industrial static bug catching and verification tools.

– Microsoft's Static Driver Verifier [1] was one of the first automatic program verification tools to apply to real-world systems code. It checks temporal safety properties of C code. It assumes memory safety and ignores concurrency, so is sound with respect to an idealized model that doesn't account for some of the programming features used in device drivers. Driver Verifier uses a whole-program analysis which would be challenging to apply incrementally, with rapid turnaround on diffs for large codebases, as INFER is at Facebook. In INFER we are only checking memory properties at present. We could check temporal properties but have not surfaced this capability to Facebook code as of yet.

– Astrée has famously proven the absence of runtime errors in Airbus code [8]. Strong soundness properties are rightfully claimed of it, for the kinds of program it targets. It also does not cover programs with dynamic allocation or concurrency, which are areas that Driver Verifier makes assumptions about. Astrée has a very accurate treatment of arithmetic, while INFER is very weak there; conversely, INFER treats dynamic allocation while Astrée does not. Astrée is a whole-program analysis which would be challenging to apply incrementally as INFER is at Facebook.
– Microsoft's Code Contracts static checker, Clousot, implements a compositional analysis by inferring preconditions in a way related to that of INFER [7]; consequently, it can operate incrementally and could likely be deployed in a similar way to INFER. Beyond this similarity, its strong points are almost the opposite of those of INFER. Clousot has a precise treatment of arithmetic and array bounds, but its soundness property is relative to strong assumptions about anti-aliasing of heap objects, where INFER contains an accurate heap analysis but is at present weak on arithmetic and array bounds. And, INFER focusses on preconditions that are *sufficient* to avoid errors, where Clousot aims for preconditions that are *necessary* rather than sufficient; necessary preconditions do not guarantee safety, but rather provide a novel means of falsification.
– Coverity Prevent has been used to find bugs in many open source and industrial programs. We are not aware of how Prevent works technically, but it has certainly processed an impressive amount of code. Coverty do not claim a soundness property, and a paper from Coverity questions whether soundness is even worthwhile [3].

5 Integration with the Development Infrastructure

Part of deploying formal verification in this environment of continuous development was the integration of INFER into the Facebook development infrastructure used by programmers. In this environment it was desirable that the programmer does not have to do anything else than his/her normal job, they should see analysis results as part of their normal workflow rather than requiring them to switch to a different tool.

At a high-level, Facebook's development process has the following phases:

1. The programmer develops a new feature or makes some change on the codebase (a.k.a. *diff*).
2. Via the source-control system, this diff goes to a phase of peer-reviews performed by other engineers. In this phase the author of the diff gets suggestions on improvement or requests for further changes from the peer reviewers.

Thus, the author and the peer reviewers start a loop of interactions aimed at making the code change robust and efficient as well as being understandable, readable and maintainable by others.
3. When the reviewers are satisfied, they “accept” the code change and the diff can be then pushed via the source-control system to the main code-base.
4. Every two weeks a version of the code base is frozen into the *release candidate*. The release candidate goes into testing period by making it available to Facebook employees for internal use. During this period, feedback from employees helps fixing bugs manifesting at runtime.
5. After two weeks of internal use, the release candidate is deployed to Facebook users. First to a small fraction of users and, if it doesn't raise any alert, it is finally deployed to all users.

During phase 2, regression tests are automatically run and before accepting any code change a reviewer requires that all the tests pass. Tests run asynchronously and the results are automatically available in the collaboration tool phabricator (`http://phabricator.org`) used for peer review.

INFER is run at phase 2. The process is completely automatic. Once the code is submitted for peer review, an analysis is run asynchronously in one of Facebook's datacenters and results are reported on phabricator in the form of comments. INFER inserts comments on the lines of code where it detects a possible bug. Moreover, we have developed tools to navigate the error trace and make it easier for the developer to inspect the bug report. To provide useful commenting on bugs we had developed a *bug hashing* system to detect in different diffs, whether two different bugs are actually the same bugs or not.

Going forward a goal is to reduce the 2 week period in step 4. There will however still remain a period here with scope for analyses and are longer-running than a per-diff analysis should be.

Incremental Analysis. On average INFER needs to comment on a diff within ten minutes, and for this the incremental analysis aspect of INFER is important. We have implemented a caching system for analysis results. The latest Android/iOS code base is fully analysed nightly. A full analysis can take over 4 hours. This analysis produces a database of pre/post-condition specifications (a cache). Using the mechanism of bi-abduction (see Section 4) this cache is then used when analysing diffs. Only functions modified by a diff and functions depending on them need to be analysed.

The Social challenge Ultimately, one of the biggest challenges we faced was a *social challenge*: to get programmers to react to bugs reported by the tool and fix genuine errors. Programmers need to accumulate trust in the analyser and they should see it as something helping them to build better software rather than something slowing them down. All the features listed in Section 3 (scalability and precision of the analysis, full automation and integration, fast reporting) are important for the social challenge.

This challenge suggested to us that we should start small to build trust gradually, and this determined our attitude on what to report. Facebook has databases of crashes and other bugs, and many on Android were out-of-memory errors and null pointer exceptions. We concentrated on these initially, targeting false positives and negatives for resource leaks and null dereferences, and we wired INFER up to the internal build process. We trained INFER first on Facebook's Android apps to improve our reports.

Having a dedicated static analysis team within Facebook helps tremendously with the social challenge.

6 Conclusions

INFER is in production at Facebook where it delivers comments on code changes submitted by developers. INFER's compositional, incremental, nature is important for this means of deployment. This stands in contrast to a model based on whole-program analysis/verification, where long runs produce bug lists that developers might fix outside of their normal workflow. We have run INFER in a whole-program mode to produce lists of issues but found this to be less effective, because of the inefficiency of the context switch that it causes when taking developers out of their flow (amongst other reasons).

Just as the apps are, INFER itself is undergoing iterative development and changing in response to developer feedback; the number of bugs reported is changing, as is the proportion of code where specs are successfully inferred. And, in addition to null dereference and leak errors, we will be extending the kinds of issues INFER reports as time goes on.

Finally, although there have been some successes, we should say that from an industrial perspective advanced program analysis techniques are generally underdeveloped. Simplistic techniques based on context insensitive pattern matching ("linters") are deployed often and do provide value, and it is highly nontrivial to determine when or where many of the ingenious ideas being proposed in the scientific literature can be deployed practically. Part of the problem, we suggest, is that academic research has focused too much on whole-program analysis, or on specify-first, both of which severely limit the number of use cases. There are of course many other relevant problem areas – error reporting, fix suggestion, precision of abstract domains, to name a few – but we believe that automatic formal verification techniques have the potential for much greater impact if compositional analyses can become better developed and understood.

References

1. Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 73–85, 2006.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 115–137, 2005.

3. Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.
4. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 459–465, 2011.
5. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
6. Josh Constine. Facebook acquires assets of UK mobile bug-checking software developer Monoidics. http://techcrunch.com/2013/07/18/facebook-monoidics.
7. Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 128–148, 2013.
8. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 21–30, 2005.
9. D.G. Feitelson, E. Frachtenberg, and K.L. Beck. Development and deployment at Facebook. *Internet Computing, IEEE*, 17(4):8–17, July 2013.
10. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, pages 1–19, 2001.
11. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.