User-Centric Static Analysis

Lisa Nguyen Quang Do
Fraunhofer IEM
lisa.nguyen@iem.fraunhofer.de

ABSTRACT

Every day, software developers produce complex and feature-rich programs. Resulting code bases are so large that single developers cannot entirely understand them, relying on tools to help them write or debug their code. One such tool is static analysis, a method of reasoning about the runtime behavior of a program at compile time to detect bugs automatically. But from code development to reporting, to testing, different uses of static analysis have different requirements for which traditional approaches are sometimes ill-adapted. This results in more work for their users, and higher abandonment rates of otherwise powerful tools. We advocate for analyses that are centered around the user's needs by introducing the concept of Just-in-Time (JIT) analyses. JIT analyses are adaptable to their use cases, reporting the most relevant results quickly, and computing the rest incrementally later. In particular, we explore the use case of code development with the JIT analysis Cheetah. We show in our experiments that Cheetah allows code developers to fix bugs twice as fast compared to equivalent traditional analyses.

1 PROBLEM & MOTIVATION

Debugging is a critical part of software development. The earlier functional and security bugs are discovered, i.e. during design or implementation, the lower the cost to fix them is [12]. According to a study conducted by the University of Cambridge [5], programmers spend 49.9% of their time debugging. One of the primary methods used to assist them in this task is static code analysis, that is, analyzing the code of a program without running it. From Google's Tricorder [21] to Oracle's Parfait [8], many companies have integrated static analysis in the development phase.

However, the adoption of static analysis tools still shows high abandonment rates [13]. Complex static analyses require considerable time and computational resources to return precise results, causing well-documented shortcomings. We have identified and aim to address three main challenges:

- **Workflow disruptions:** According to the size of the analyzed code base, an analysis can take hours to complete, which makes the use of code analysis in an Integrated Development Environment (IDE) difficult. This forces programmers to separate the tasks of coding and debugging. Previous studies have shown that static analysis tools should not disrupt the developer's workflow [7, 13], making them more likely to be adopted [24].
- **False positives:** Static analysis tend to report many warnings, among which many are discarded by the users [3].
- **Analysis customization:** Because users don't know how an analysis operates internally, it is sometimes difficult for them to determine why certain warnings are reported, and which ones to address in priority. They pour over long lists of warnings, and put considerable effort into eliminating false positives and fixing bugs efficiently [8, 13].

While static analysis tools can be very powerful, their potential is wasted when they are not adapted to their use case. Traditional static analyses work independent of the user: users have little knowledge and control over how an analysis works internally, and the analysis does not take into account user knowledge about the analyzed code. In the case of code development, the use of static analysis can break the code developer's workflow and add an overhead to their work. In addition, the analysis does not capitalize on the guidance the developer can provide to simplify its task (e.g. focus on certain parts of the code).

In order to provide better support for debugging software -especially during code development, we advocate for more adaptive, user-centric static analyses. In our work, we explain how a large class of static analyses can be adapted to take user requirements into account during the analysis in order to return in priority those results that are most interesting to the user. Interesting results can be results that are easier to fix, faster to compute, or that are more likely to be true positives for example.

In this work, we make the following contributions:

- We introduce the concept of Just-in-Time (JIT) analysis [16] that allows static analysis writers to specify prioritization properties used to direct the analysis towards those results that are most interesting to the end-user.
- We instantiate this concept through a layered analysis system, and show that existing static analyses can be adapted to support it with minimal changes.
- We implement Cheetah, a JIT analysis that detects privacy leaks in Android applications. Focusing on the use case of code development, Cheetah returns in priority those results that are located around the code developer's working set, gradually expanding the analysis scope to encompass methods, classes, and modules further away. Early computations produce simple results quickly enough for Cheetah to be integrated in an IDE.
- We evaluate Cheetah, focusing on performance and developer experience. Our experiments show that Cheetah is able to return initial results under a second, and that using Cheetah in the IDE enables code developers to fix warnings twice as fast as with a traditional analysis.

We refer the reader to our technical report [6] for the JIT algorithm, its proofs of soundness and termination, the details of Cheetah's implementation, and the setup and raw data of our empirical evaluation and user study. Cheetah is open-sourced [6] and available under the EPL license.

```
1  public class A {
2    void foo(B b)
3      String s = getPwd();
4      String t = s;
5      String u = s;
6      b.leakPwd(t);
7      leakPwd(s);
8      log(u);//privacy leak (A)
9    }
10   void leakPwd(String x) {
11     log(x);//privacy leak (B)
12   }
13 }
14 public class B {
15   void leakPwd(String y) {
16     log(y);//privacy leak (C)
17   }
18 }
```

Figure 1: CWE-200: Information Exposure.

```
19 void encrypt(Y y, Z z) {
20   Cipher g = new Cipher();
21   z.maybeInit(g);
22         // polymorphic call
23   g.doWork(); (D)
24
25   Cipher h = new Cipher();
26   y.maybeInit(h);
27         // monomorphic call
28   h.doWork();  (E)
29 }
30
31 // class X extends Z
32 void maybeInit(Cipher a) {
33   a.init();
34 }
35
36 // class Y extends Z
37 void maybeInit(Cipher b) {
38 }
```

Figure 2: CWE-227: API Abuse.

```
39 void main() {
40    F g = new F();
41    F h = new F();
42    F f = null;
43
44    g = f;
45
46    if(...) h = f;
47
48    x = f.a; (F)
49    y = g.a; (G)
50    z = h.a; (H)
51 }
```

Figure 3: CWE-476: Null Pointer Dereference.

2 BACKGROUND & RELATED WORK

2.1 Data-Flow Analysis

Figures 1-3 present eight bugs/security vulnerabilities, marked as (A) to (H). Figure 1 contains three privacy leaks, with the password retrieved at line 3 being written to a log at lines 8, 11, and 16. Figure 2 contains bad initializations of `Cipher`. Objects of type `Cipher` must be initialized with a call to `init()` before they are used. In the example, `h` is incorrectly initialized, and `g` may be. In Figure 3, `f` is set as null at line 42, `g` and `f` are aliased at line 44, and `h` and `f` may be aliased at line 46. This causes potential null pointer dereferences at lines 48, 49, and 50.

All eight bugs can be detected using static data-flow analysis. Data-flow analysis is a class of static analysis that tracks values along the different statements of a program. For example, to detect (F), (G) and (H), a data-flow analysis would track statement by statement whether the different objects of the program are `null` or not. Then, if it encounters a pointer access to an object known as null, it would report it. Similarly, an analysis tracking sensitive data-flows from *sources* to *sinks* would detect privacy leaks. In Figure 1, the source is at line 3, and the three sinks, at lines 8, 11, 16. Such an analysis is called a *taint* analysis [22].

2.2 Batch-Style Analysis

Traditional static analyses typically compute results over the whole program space, before reporting them all at once. In this paper, we will refer to this behavior as *batch-style*. To avoid overwhelming the users with too many results, analysis tools often provide a classification system, grouping results by confidence for example. Notable ordering and classification approaches [17] include mining software history to determine typical fixes [14], machine learning on result features [15], or querying the user [9].

Batch-style tools apply those techniques as post-processing modules, run after the analysis terminates. On the other hand, JIT analyses compute results depending on ordering properties at analysis time, delivering results directly in the right order. This guarantees that the first results are reported immediately, and that they are of most interest to the user. Applied to code development, this allows a JIT analysis to run in the background of an IDE, as the developer writes. This system is much less intrusive, as it does not need to interrupt the user while the analysis runs, and returns results in small, digestible batches instead of a single long list.

2.3 Incremental Analysis

Incremental analyses [1, 25] operate on small changes to the code base. For each change made by the user, the analysis only recomputes those results that are affected by the change. Like CHEETAH, such approaches efficiently shorten the long waiting times. But they require regular full-program analysis runs from time to time in order to re-initialize the state of the analysis. They also do not provide any inherent result ordering.

2.4 Staged Analysis

Like CHEETAH, Parfait [8] operates in stages. It runs an initial bug detector, and cascades different analyses in an increasing order of complexity and a decreasing order of efficiency to confirm the initial findings. Unlike our approach, one analysis layer in Parfait may invalidate bugs reported by a previous layer. Results that appear and disappear later can be confusing for use cases where results are reported to humans as they are found. In our approach, each layer of a JIT analysis uses previously computed information to detect *new* bugs and does not invalidate previously reported warnings. This minimises disruption in the developer workflow.

3 APPROACH

3.1 Just-in-Time Analysis

In order to report all potential issues, a static analysis must explore every possible run scenario of the analyzed program. In Figure 2, the call at line 21 may dispatch to both `maybeInit()` methods, so the analysis must model both possibilities.

A batch-style analysis may visit those possibilities in any order. A JIT analysis prioritizes some possibilities over others, based on user-defined ordering properties. The analysis writer specifies in the analysis which parts of the code should be analyzed first, and which ones should be left for later. In essence, a JIT analysis gives the user control over the analysis, and allows them to guide it to suit their use case.

Let us consider the case of code development. The end-users (e.g. developers) are more likely to be interested in warnings that are located around their current working set. By prioritizing local flows, an analysis can explore a smaller code base first, and gradually increase the analysis scope to include methods, classes, etc. Unlike batch-style analyses which typically start at the `main` method, a JIT analysis can start at any point in the code. Here, the analysis' *starting point* is the current edit point. Supposing that the developer's current edit point is located in method `foo` in Figure 1, a JIT taint analysis would report (A) first -as it is located in `foo`, then, (B) -as it is in the same class, and finally, (C).

With another ordering system based on computational resources, an analysis could for example resolve monomorphic calls before polymorphic calls, which are more computationally expensive to analyze since all potential callees must be explored. Thus, (E) would be reported first and (D) would be left for later. Similarly, resolving aliases also requires extra computation, so (F) would be returned before (G) and (H).

Another prioritization example is by confidence, returning those results that are most likely to be true positives first. Monomorphic calls more likely lead to true positives than polymorphic calls. Similarly, direct assignments (line 42) can be prioritized over conditional assignments (line 46).

3.2 Layering System

The prioritization system of a JIT analysis is based on priority *layers*. We define *trigger* points in the analyzed program as statements at which the analysis pauses and comes back later. The priority layers determine when the analysis resumes at a particular trigger point compared to others.

Let us consider a taint analysis analyzing the program of Figure 1. For code development, we consider a prioritization system by locality, as presented in Section 3.1, where users are most interested in those results that are close to their working set. The analysis' starting point is the current edit point. The triggers are method call sites, making the analysis pause and resume at lines 7 and 6. The priority layers prioritize call sites by distance to the current edit point, as shown in Table 1. Supposing that the starting point of the analysis is `foo`, the trigger at line 7 will have a higher priority as the one at line 6, as it refers to a method in the same class as `foo`, as opposed to the same file.

Layer	Distance from the edit point
L1: Method	Same method.
L2: Class	Same class.
L3: Class Lifecycle	Lifecycle methods in the same class.
L4: File	Same file.
L5: Package	Same package.
L6: Monomorphic	In the project, along monomorphic calls.
L7: Polymorphic	In the project, along polymorphic calls.
L8: Android Lifecycle	In the project, along the implicit dataflows in lifecycle methods, to handle interactions between various application components.

Table 1: Layers of CHEETAH. Each layer extends the analysis to a certain distance from the edit point.

The resulting JIT taint analysis executes as follows:

(1) The user triggers the analysis at the `foo` method.
(2) The analysis starts analyzing the `foo` method.
  (a) The analysis pauses at line 6. It ignores the call and continues analyzing `foo`.
  (b) The analysis pauses at line 7. It ignores the call and continues analyzing `foo`.
  (c) (A) is found and reported.
(3) The analysis resumes at line 7 (because its priority is higher than the call at line 6). It explores the call at line 10 and reports (B).
(4) The analysis resumes at line 6. It explores the call at line 15 and reports (C).

3.3 Architecture of the JIT Framework

A traditional batch-style data-flow analysis is generally implemented on top of an analysis solver, i.e. the engine that runs the analysis. When implementing an analysis (e.g. a taint analysis), the analysis writer must specify how the analysis should track which values throughout a program by overwriting *transfer functions*.

In the case of a JIT analysis, the solver contains an additional prioritization module that handles the pause-resume behavior of the analysis based on the specified triggers and layers. In addition to the transfer functions, the writer must also specify (1) the starting point(s) of the analysis, (2) the different layers, (3) how to recognize a trigger and (4) how the priority layers map to the triggers.

With this architecture, an analysis writer can instantiate different types of analyses and different prioritization policies without modifying the JIT solver. It is also possible to modify existing batch-style analysis solvers to add a prioritization module, transforming existing batch-style analyses into JIT analyses with minimal changes, as shown in our technical report [6].

4 CHEETAH

We have implemented CHEETAH, a JIT taint analysis that detects privacy leaks in Android applications. CHEETAH is designed for code development, and aims at providing results close to the current edit point in priority. It defines trigger points as call sites, and follows the layering system presented in Table 1. This choice of layers also helps return the first results quickly, as lower layers (**L1**-**L3**) require minimal class loading and computational resources.

Dedicated layers (**L3** and **L8**) help model the callbacks of the Android framework. Similarly to Arzt et al. [2], we create a `dummyMain` method that makes the method calls between those callbacks explicit. In our approach, the `dummyMain` is distributed over the layering system as well.

To support code developers, CHEETAH is designed to analyze the whole code base, including unreachable code. When writing an application, developers may work on unreleased features or incomplete code. Typical analyses ignore unreachable code, a feature that CHEETAH provides.

CHEETAH is built on top of the Soot analysis framework [23] and the Heros IFDS solver [4]. It uses the sources and sinks definitions described by Rasthofer et al. [19]. CHEETAH is integrated in the Eclipse IDE as a plugin (Figure 4) and runs in the background as the developer codes. It is triggered every time the project is built, starting from the method that has the focus. A demo video is available online [6].

5 RESULTS

To evaluate how CHEETAH integrates into the development workflow, we conducted a set of empirical experiments and a user study. We compared CHEETAH to BASE, a batch-style taint analysis for Android applications. We first implemented BASE and modified it as described in Section 3.3 to obtain CHEETAH. Thus, both analyses share the same solver and transfer functions. We ran our experiments on a 64-bit Windows 7 machine with one dual-core Intel Core i7 2.6 GHz CPU running Java 1.8.0_102.

5.1 Empirical Evaluation

CHEETAH and BASE were run on 14 real-life Android applications from F-Droid [11]. We measured when each warning was reported. We conducted two sets of experiments with different starting points for CHEETAH. The first set was run with starting points close to known privacy leaks, representing cases when the user investigates a particular bug. The second set was run with 20 starting points per application, randomly selected using Boa [10]. This represents cases when users are not using CHEETAH when developing. BASE has a unique starting point: a dummy main method [2].

The experiments show that:

- CHEETAH reports the first result in a median time of less than one second (Nielsen's recommended threshold for interactive user interfaces [18]), allowing the developer to remain focused on their work.
- CHEETAH reports 11% of its warnings in lower layers (**L1**-**L4**). If directed to known sources of bugs (starting points near known privacy leaks), CHEETAH reports 33% of its warnings in layers **L1**-**L4**. We see that when the user focuses on a particular warning, the analysis reports the first warnings faster, in earlier layers.
- CHEETAH returns the last result in 4.26× more time than BASE. We attribute this to the full code coverage feature of CHEETAH, which considers a much larger code base than BASE for the same application.

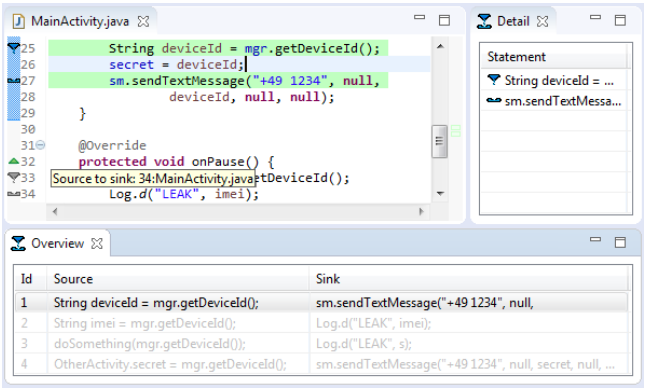

Figure 4: Graphical user interface of CHEETAH. The Overview view (bottom) lists the warnings as they are reported. The Detail view (right) presents the details of a selected leak. All warnings in the Overview view are grayed out at each new run of the analysis. They switch to black when they are re-confirmed. Fixed warnings are removed. The source-sink icons in the left gutter are grayed out accordingly.

5.2 User Study

We integrated both CHEETAH and BASE as Eclipse plugins, and conducted a user study with 17 participants of diverse backgrounds and knowledge of static analysis. The participants were given the task of refactoring Bites, an Android application from F-Droid [11], while keeping the number of privacy leaks to a minimum. Each participant performed the task twice on different parts of the application: once with CHEETAH and once with BASE. Nine participants started with CHEETAH, eight with BASE. In the time limit of 10 minutes allocated to each task, we measured how many leaks the participants fixed, and how long it took to fix them. Afterwards, the participants filled a questionnaire and were interviewed individually.

We have observed the following:

Tasks: Figure 5 presents the distribution of the time taken to fix a leak for the two tasks. We see that for both tasks, participants using CHEETAH took half as long as participants using BASE to fix a leak. We also observed that participants using CHEETAH fixed more leaks than participants using BASE. For task 1, CHEETAH users fixed a median of 4 leaks and BASE users fixed 2 leaks. For task 2, CHEETAH users fixed 4 leaks, and BASE users, 3 leaks.

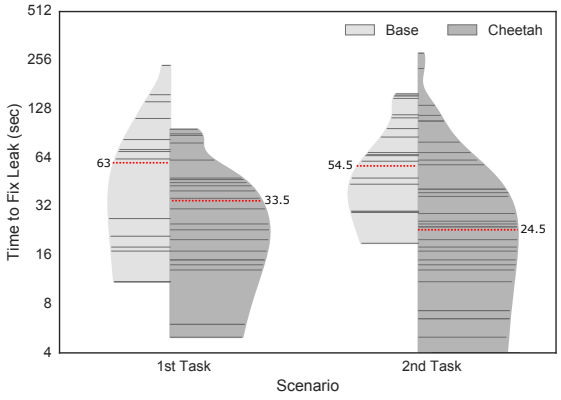

Figure 5: Violin plot representing the distribution of the times to fix leaks across all participants, by task and tool used during the task. Each horizontal line represents data leaks fixed in the corresponding time. The length of a line is the number of leaks fixed in that time. Dashed lines represent medians.

Questionnaire: In the questionnaire, participants were asked to compare both tools. When asked about the likelihood of recommending the tools to a friend on a 11-point Likert scale [20], CHEETAH's score was of 7.4, against 2.7 for BASE. Participants found CHEETAH less complex than BASE. They found its functionalities well-integrated, and responded that they were more likely to use it frequently compared to BASE.

Interviews: Twelve participants reported that CHEETAH was best suited for code development, due to its quick updates. Two participants expressed concerns on CPU usage, as CHEETAH performs more work than BASE on the same application.

6 LIMITATIONS AND FUTURE WORK

We have presented the concept of JIT analysis, which aims at integrating user requirements into static analysis. We have seen how to derive a JIT analysis from a batch-style analysis, and have shown with CHEETAH that user-centric analyses can provide better user support than traditional analyses.

As it is now, CHEETAH performs a whole-program analysis at each run, re-computing unnecessary information. We plan to incrementalize it in the future. The JIT algorithm is currently limited to distributive data-flow analyses. It would be interesting to extend it to other types of analyses.

CHEETAH is one application of JIT analyses for code development. In the future, we plan to develop a more general framework to make batch-style analyses user-centric. This would simplify the definition of layering systems, allowing analysis writers to implement JIT analyses for different use cases, and produce more useful static analyses.

ACKNOWLEDGEMENTS

This work is a collaboration between Lisa Nguyen Quang Do, Karim Ali, Eric Bodden, Benjamin Livshits, Emerson Murphy-Hill, and Justin Smith.

REFERENCES

[1] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *ICSE'14*.
[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI'14*.
[3] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* (2010).
[4] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP'12*.
[5] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2013. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers. (2013).
[6] Cheetah. 2017. https://blogs.uni-paderborn.de/sse/tools/cheetah-just-in-time-analysis/. (2017).
[7] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. (2016).
[8] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. 2012. Transitioning Parfait into a Development Tool. *IEEE Security & Privacy* (2012).
[9] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *PLDI'12*.
[10] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE'13*.
[11] F-Droid. 2017. Free and Open Source Android App Repository. https://f-droid.org. (2017).
[12] Research Triangle Institute. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. (2002).
[13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *ICSE'13*.
[14] Sunghun Kim and Michael D. Ernst. 2007. Prioritizing Warning Categories by Analyzing Software History. In *MSR'07*.
[15] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. 2010. Automatic Construction of an Effective Training Set for Prioritizing Static Analysis Warnings. In *ASE'10*.
[16] Nguyen Quang Do Lisa, Ali Karim, Livshits Benjamin, Bodden Eric, Smith Justin, and Murphy-Hill Emerson. 2017. Cheetah: Just-in-Time Taint Analysis for Android Apps. In *ICSE'17-Demonstrations*.
[17] Tukaram Muske and Alexander Serebrenik. 2016. Survey of Approaches for Handling Static Analysis Alarms. In *SCAM'16*.
[18] Jakob Nielsen. 1994. *Usability Engineering*. Elsevier.
[19] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS'14*.
[20] Frederick F Reichheld. 2003. The one number you need to grow. *Harvard Business Review* (2003).
[21] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *ICSE'15*.
[22] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *PLDI'09*.
[23] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *CC*.
[24] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *FSE'15*.
[25] Jing Xie, Heather Lipford, and Bei-Tseng Chu. 2012. Evaluating Interactive Support for Secure Programming. In *CHI'12*.