Cheetah: Just-in-Time Taint Analysis for Android Apps

Lisa Nguyen Quang Do*, Karim Ali†, Benjamin Livshits‡, Eric Bodden§, Justin Smith¶, and Emerson Murphy-Hill¶
*Fraunhofer IEM, Germany, lisa.nguyen@iem.fraunhofer.de
†University of Alberta, Canada, karim.ali@ualberta.ca
‡Imperial College London, United Kingdom, b.livshits@imperial.ac.uk
§Heinz Nixdorf Institute at Paderborn University & Fraunhofer IEM, Germany, eric.bodden@upb.de
¶North Carolina State University, USA, jssmit11@ncsu.edu and emerson@csc.ncsu.edu

***Abstract*—Current static-analysis tools are often long-running, which causes them to be sidelined into nightly build checks. As a result, developers rarely use such tools to detect bugs when writing code, because they disrupt their workflow. In this paper, we present Cheetah, a static taint analysis tool for Android apps that interleaves bug fixing and code development in the Eclipse integrated development environment. Cheetah is based on the novel concept of Just-in-Time static analysis that discovers and reports the most relevant results to the developer fast, and computes the more complex results incrementally later. Unlike traditional batch-style static-analysis tools, Cheetah causes minimal disruption to the developer's workflow. This video demo showcases the main features of Cheetah:** `https://www.youtube.com/watch?v=i_KQD-GTBdA`.

I. Introduction

Integrating static analysis in the development process enables early detection of software bugs, which reduces the cost of fixing them. However, most developers do not use static-analysis tools, because they generate many false positives and may take hours, or even days, to run on sizeable code bases [1], [2]. Therefore, most companies sideline static-analysis tools, such as PREfix [3], PREfast [4], Fortify [5], and Coverity [6], to nightly build checks to avoid disrupting the workflow of software developers.

We argue that interleaving code development and bug fixing enables a less disruptive and more usable integration of static analyses into development environments (IDEs), similar to, for example, the incremental Java compiler in Eclipse. To achieve that, we introduce the novel concept of Just-in-Time (JIT) static analysis that takes into account the development context to report relevant, easy-to-fix results fast. The approach depends on the general concept of *layering*, such that initial analysis layers narrow the analysis scope to provide quick and relevant warnings, and later layers compute more complex results while the developer handles the first ones. In this paper, we introduce Cheetah, an Eclipse plugin that implements a JIT taint analysis for Android apps. Cheetah is designed to help software developers detect insecure information flows in their applications within seconds, with minimal disruption to their workflow. In particular, Cheetah supports software developers by providing the following features:

- seamless integration in Eclipse,
- reporting warnings using decluttered views,
- ensuring the validity of each warning,
- covering the full codebase,
- highlighting stale analysis information, and
- providing quick feedback on a particular warning.

II. Overview of Cheetah

A JIT analysis uses the notion of layering to gradually perform computations by expanding its scope, which allows it to quickly report its first results. On the other hand, traditional batch-style analyses typically order results using a post-processing module after the analysis is done [7]. Our JIT framework [8] enables easy transformation of a distributive dataflow analysis [9] to its corresponding JIT with minimal changes to its transfer functions. A JIT analysis computes the same dataflow propagations as its base analysis, but it delays some propagations in favor of others by pausing and resuming them later at *trigger* statements. Each trigger is associated with a priority that determines the layer at which the JIT analysis resumes its computation. This approach enables a JIT analysis to run in the background of the IDE, returning few warnings at a time and computing further results while the developer processes the initial ones. This mechanism of computing the analysis results avoids the *wall of bugs* effect [1] and reduces the *perceived* analysis latency, which helps improve the overall usability of a JIT-based analysis tool.

Table I presents the analysis layers in Cheetah, where triggers are method calls. Cheetah runs on the initial layers that are closer to the current edit point of the developer, and resolves more distant method calls in later layers. This layering by *locality* enables Cheetah to return more relevant results first. Those results are typically closer to the current context of the developer and are easier to understand and fix. Initial results typically have shorter traces and are more likely to be true positives compared to, for example, results that Cheetah reports for polymorphic calls in **L7**. Other mechanisms can be used to define the layers of a JIT analysis.

TABLE I: The analysis layers in Cheetah.

Layer	Name	Description
L1	Method	Cheetah propagates the dataflows in the same method as the current edit point.
L2	Class	Cheetah propagates the dataflows along calls to methods in the same class as the current edit point.
L3	Class Lifecycle	Cheetah propagates the dataflows in lifecycle methods in the same class as the current edit point.
L4	File	Cheetah propagates the dataflows along calls to methods in the same file as the current edit point.
L5	Package	Cheetah propagates the dataflows along calls to methods in the same package as the current edit point.
L6	Project Monomorphic	Cheetah propagates the dataflows along the monomorphic calls in the project.
L7	Project Polymorphic	Cheetah propagates the dataflows along the polymorphic calls in the project.
L8	Android Lifecycle	Cheetah propagates the implicit dataflows in lifecycle methods in the whole project.

```
1  public class A {
2    void main(C b)
3      s.f = secret(); // source
4      t = s;
5      if(...) u = s;
6      sendMessage(u);
7      b.sendMessage(t);
8      leak(s.f); // sink (A)
9    }
10   void sendMessage(String x) {
11     leak(x.f); // sink (B)
12   }
13 }
14 public class B extends C {
15   void sendMessage(String y) {
16     leak(y.f); // sink (C)
17   }
18 }
```

Fig. 1: An example illustrating the workflow of Cheetah.

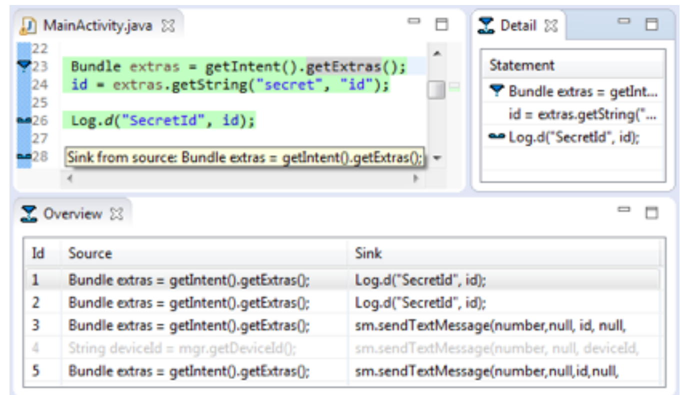

Fig. 2: The graphical user interface of Cheetah.

III. Running Example

A taint analysis tracks sensitive dataflows from sources to sinks to either detect privacy leaks [10]. The program in Fig. 1 contains three privacy leaks from the source (line 3) to the sinks (A) (line 8), (B) (line 11), and (C) (line 16). While a batch-style analysis computes all three warnings in no particular order, Cheetah prioritizes them according to their proximity to the current edit point. If the developer edits `main`, Cheetah first reports (A), because it is local to `main`, then (B), because it is further away from the edit point (in the same class but not the same method), and finally (C), because it is located in a different class.

IV. Main Features of Cheetah

We built Cheetah on top of the Soot analysis framework [11] and the Heros IFDS solver [12]. Fig. 2 highlights the main features of Cheetah.

Seamless Integration with Eclipse. To enable a smooth integration with Eclipse, Cheetah hooks into the Eclipse incremental builder. Whenever the project is saved, Cheetah starts running at the method that currently holds the focus. Every run of Cheetah kills any previous analysis instances, invalidating previous results until the current instance of Cheetah confirms or refutes them.

Decluttered Views. In early prototypes, users complained that Cheetah showed all warnings in one single view, so we redesigned the user interface to show the warnings in two separate views to declutter the reported information. The *Overview* view provides a list of all reported warnings, and the *Detail* view displays a trace of the selected warning. This trace represents an evidence that there exist a path from the source to the sink, ensuring the validity of the reported warning.

Full Code Coverage. Unlike traditional analyses, Cheetah aims at supporting software developers who may be working on unreleased features that are not reachable yet in the codebase. While traditional analyses typically ignore unreachable code, Cheetah analyzes the full codebase, including unreachable code, to report warnings about those unreleased features to the developer. This is a useful feature for development scenarios where developers work on incomplete programs or programs that may mot even have a main method.

Color-coded Warnings. In early prototypes, users found it confusing that Cheetah removed, and possibly re-ordered, the previous results in the Overview view. Therefore, we redesigned Cheetah to use different colors to indicate the state of each warning. A warning in Cheetah can be: *active* (confirmed by the latest Cheetah run),

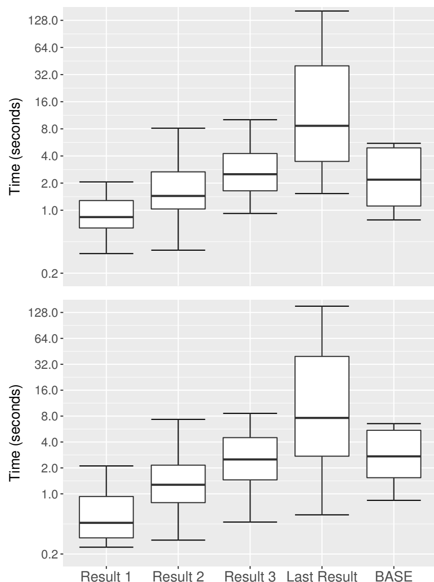

Fig. 3: Time to report results (in log scale) for Cheetah and Batch, starting at SPB (top) and SPS (bottom).

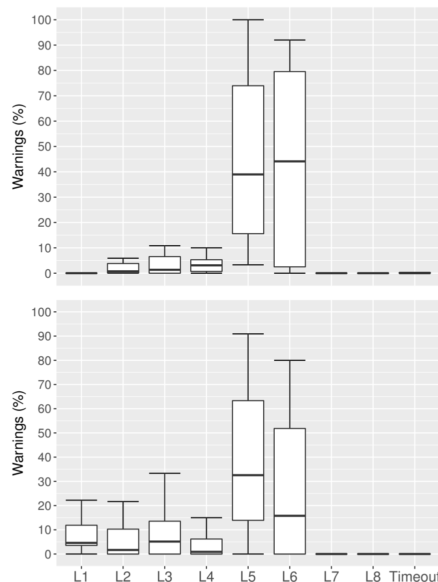

Fig. 4: Percentage of warnings reported at each layer in Cheetah with SPB (top) and SPS (bottom).

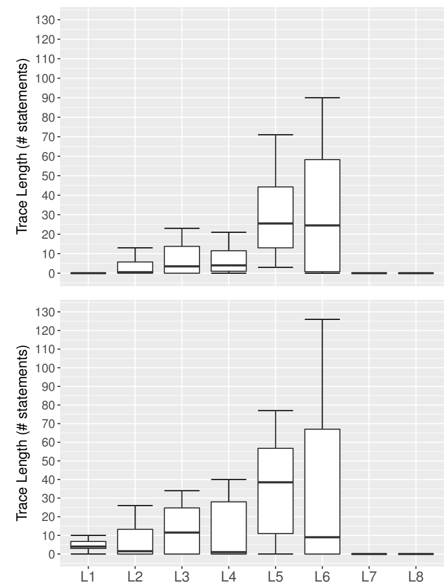

Fig. 5: Trace length of the warnings reported at each layer in Cheetah with SPB (top) and SPS (bottom).

pending (found by a previous run of Cheetah but not yet confirmed or refuted by the latest run), or *fixed*. Cheetah shows active warnings in black and pending warnings in gray. Fixed warnings are grayed out during the current run, and are removed the next time Cheetah runs, allowing developers to quickly check if their fix was effective.

Descriptive Icons. Similar to icons for compilation errors in Eclipse, Cheetah adds source and sink icons to the left gutter. Cheetah grays out those icons if the selected warning is pending, providing quick feedback to the developer. Cheetah also uses tooltips provide additional information about each statement in the trace of the selected warning.

Other features. To clarify the presentation of its results, Cheetah highlights the trace of the selected warning in the code. Additionally, Cheetah uses unique identifiers to help users keep track of the reported warnings.

V. Evaluation

We empirically evaluate Cheetah by comparing it to its traditional batch-style counterpart (referred to as Batch). We ran all our experiments on a 64-bit Windows 7 laptop with a dual-core 2.6 GHz Intel Core i7 CPU running Java 1.8.0-102, and limited the Java heap space to 1 GB.

A. Benchmark Evaluation

To compare the performance of Cheetah and Batch, we use a benchmark suite of 14 real-world Android apps from F-Droid [13]. We ran two experiments for each app. In the first experiment, Cheetah starts at 20 randomly selected methods that we collected using Boa [14] (referred to as SPB). In the second experiment, Cheetah starts at sources of known data leaks (referred to as SPS). While SPS represent cases when the user is investigating a particular bug, SPB represent cases when the user is not necessarily using Cheetah during code development. Batch has one starting point: a dummy main method that acts as the entry point to the Android app [10].

Fig. 3 shows that Cheetah reports the first result in a median time less than Nielsen's 1 second recommended threshold for interactive user interfaces [15]. However, Cheetah takes more time than Batch to report all warnings, because, unlike Batch, it analyzes the full codebase. Fig. 4 shows that when Cheetah starts at SPS, it reports more warnings at the initial layers, especially **L1** and **L3**. For those cases, Fig. 5 shows that the median length of the reported traces is less than 11 statements, making them easier to interpret. Across all apps, Cheetah has less than 1% timeouts, due to the process of computing the trace.

B. User Study

To compare the integration of Cheetah into the workflow of developers to that of Batch, we conducted a user study that involves 18 participants, half of which are professional developers. Given 10 minutes, the participants had to remove code duplicates in a real-life Android application from F-Droid [13], while minimizing the number of reported warnings. The participants then responded to 29 questions to assess the merits of both tools. We then interviewed them to provide further details about their experience using Cheetah and Batch.

Measuring usability using the aggregated System Usability Score (SUS) [16], 12 participants found Cheetah more usable for code development than Batch. Using a two-tailed Wilcoxon Signed-Rank test [17] ($p < 0.05$), compared to Batch, participants less likely found that Cheetah is unnecessarily complex or cumbersome (-0.6 mean response). Additionally, participants are more likely

to use Cheetah frequently (+0.7 mean response), and found its functions well-integrated (+0.5 mean response). Nevertheless, two participants raised concerns about CPU overhead, because Cheetah re-analyzes the whole program at each save, cancelling the previous run of the analysis. We plan to overcome this issue by transforming Cheetah into an incremental analysis.

VI. Related Work

Significant work has been done to make static analyses more responsive. For example, Solstice [18] runs an offline analysis on a replica of the developer's workspace, and reports results in a non-disruptive manner. In contrast, Cheetah is an interactive analysis that operates on the original codebase, reporting its results in a timely fashion.

Incremental analyses, such as Reviser [19], ASIDE [20] and ECHO [21], re-analyze only the recent code changes and updates the relevant analysis results. While Cheetah re-analyzes the whole program at every run, it consistently reports its first results fast, unlike incremental analyses whose first run is typically slower. Additionally, Cheetah returns its results in a specific order. We plan to incrementalize Cheetah to leverage the best of both approaches.

Parfait [7] runs different analyses in layers of increasing order of complexity and decreasing order of efficiency. Unlike Parfait, Cheetah layers a single analysis, making it more responsive in general. Moreover, later analyses in Parfait may invalidate the results that the initial analyses have already reported. On the other hand, later layers in Cheetah do not refute the results that have been reported by earlier analysis layers.

VII. Conclusion

We have presented Cheetah, a JIT taint analysis tool for Android apps that interleaves code development and static analysis execution. Our empirical results show that Cheetah helps software developer detect data leaks, with minimal disruption to their workflow. The setup and raw data of our experimental results and user study is available online, as well as a technical report detailing the tool's implementation [8]. Cheetah is open-sourced [8] and available under the EPL license [22].

Acknowledgments

This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation. This material is also based upon work supported by the National Science Foundation under grant number 1318323.

References

[1] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[2] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *International Conference on Automated Software Engineering (ASE)*, pages 332–343, 2016.

[3] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice & Experience (SPE)*, 30(7):775–802, 2000.

[4] PREfast. `https://msdn.microsoft.com/en-us/library/ms933794.aspx`.

[5] HP Fortify. `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/`.

[6] Coverity. `http://www.coverity.com/`.

[7] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. Transitioning Parfait into a Development Tool. *IEEE Security & Privacy*, 10(3):16–23, 2012.

[8] Cheetah. `https://blogs.uni-paderborn.de/sse/tools/cheetah-just-in-time-analysis/`.

[9] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[11] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Compiler Construction (CC)*, pages 18–34, 2000.

[12] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *International Workshop on State of the Art in Java Program Analysis (SOAP)*, pages 3–8, 2012.

[13] F-Droid. Free and Open Source Android App Repository. `https://f-droid.org`.

[14] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, pages 422–431, 2013.

[15] Jakob Nielsen. *Usability Engineering*. Elsevier, 1994.

[16] John Brooke et al. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[17] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[18] Kivanç Muslu, Yuriy Brun, Michael D. Ernst, and David Notkin. Making offline analyses continuous. In *Foundations of Software Engineering (FSE)*, pages 323–333, 2013.

[19] Steven Arzt and Eric Bodden. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *International Conference on Software Engineering (ICSE)*, pages 288–298, 2014.

[20] Jing Xie, Heather Richter Lipford, and Bei-tseng Chu. Evaluating interactive support for secure programming. In *Conference on Human Factors in Computing Systems (CHI)*, pages 2707–2716, 2012.

[21] Sheng Zhan and Jeff Huang. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Foundations of Software Engineering (FSE)*, pages 775–786, 2016 (to appear).

[22] Eclipse Public Licence. `https://eclipse.org/legal/eplfaq.php`.