Just-in-Time Static Analysis

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

Karim Ali
University of Alberta, Canada
karim.ali@ualberta.ca

Benjamin Livshits
Imperial College London, U.K.
livshits@ic.ac.uk

Eric Bodden
Paderborn University and Fraunhofer IEM, Germany
eric.bodden@upb.de

Justin Smith
North Carolina State University, USA
jssmit11@ncsu.edu

Emerson Murphy-Hill
North Carolina State University, USA
emerson@csc.ncsu.edu

ABSTRACT

We present the concept of Just-In-Time (JIT) static analysis that interleaves code development and bug fixing in an integrated development environment. Unlike traditional batch-style analysis tools, a JIT analysis tool presents warnings to code developers over time, providing the most relevant results quickly, and computing less relevant results incrementally later. In this paper, we describe general guidelines for designing JIT analyses. We also present a general recipe for transforming static data-flow analyses to JIT analyses through a concept of layered analysis execution. We illustrate this transformation through Cheetah, a JIT taint analysis for Android applications. Our empirical evaluation of Cheetah on real-world applications shows that our approach returns warnings quickly enough to avoid disrupting the normal workflow of developers. This result is confirmed by our user study, in which developers fixed data leaks twice as fast when using Cheetah compared to an equivalent batch-style analysis.

CCS CONCEPTS

•Security and privacy → Software security engineering; •Software and its engineering → Software verification;

KEYWORDS

Static analysis, Just-in-Time, Layered analysis

ACM Reference format:
Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-Time Static Analysis. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092705

1 INTRODUCTION

More companies are integrating static analysis checks in their development process to detect software bugs early in the development lifecycle. However, most static analysis tools, such as Microsoft's PREfix/PREfast [9, 32], HP Fortify [19], and Coverity [14], are designed to be used in batch mode, because analyzing real-life projects can easily take hours. Therefore, many companies run static analysis tools at major release points in the product lifecycle or as part of nightly builds. In those use cases, developers pour over long lists of warnings (often in the order of thousands of warnings for real-life projects), deciding which messages correspond to real errors that require a fix [4, 22, 26, 38]. Running static analysis tools in this batch mode limits the potential utility of static analysis: by the time the results are generated, the developer may have forgotten the coding context to which these results pertain.

In this paper, we propose the Just-in-Time (JIT) static analysis concept, in which we advocate for the integration of static analysis into the development workflow, allowing developers to immediately see the impact of their changes in the code without blocking them from performing other coding tasks. Integrating the analysis into the development environment (IDE) enables reporting more manageable, "digestible" sets of warnings almost continuously, instead of providing the user with a long list of warnings at the end of the analysis run. We also advocate for a different delivery strategy for the results: return simple-to-fix and more precise results first and use the time that the developers take to address them to compute more complex, false-positive-prone results later, while integrating developer feedback.

We instantiate this concept through a layered analysis, which starts at the program point currently edited by the developer, gradually expanding the analysis scope to encompass methods, classes, files, and modules further away. Early analysis layers quickly produce intra-procedural results, known to yield few false positives. Later layers may find more complex results further out, but also run the risks of higher analysis running times and more false positives.

To concretely illustrate the concept of JIT static analysis, we instantiate the layered JIT analysis framework with Cheetah, a JIT taint analysis for Android applications, that we have made publicly available [10]. To evaluate the benefits and shortcomings of our JIT approach compared to a traditional batch-style analysis, we have conducted an empirical evaluation of Cheetah using real-world Android applications, as well as a user study with 18 participants.

This paper makes the following contributions:

- It proposes the concept of JIT analysis that interleaves the process of computing analysis warnings with that of the developer fixing them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA'17, Santa Barbara, CA, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

978-1-4503-5076-1/17/07…$15.00
DOI: 10.1145/3092703.3092705

```
1   public class A {
2     void main(B b)
3       String s = secret(); // source
4       String t = s;
5       String u = s;
6       sendMessage(s);
7       b.sendMessage(t);
8       leak(u); // sink, leak Ⓐ
9     }
10    void sendMessage(String x) {
11      x = ''not tainted'';
12      leak(x); // sink, no leak
13    }
14  }
15  public class B {
16    void sendMessage(String y) {
17      leak(y); // sink, leak Ⓑ
18    }
19  }
```

Figure 1: Running example illustrating a JIT taint analysis.

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

Figure 2: Example illustrating JIT API misuse detection.

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

Figure 3: Example illustrating a JIT nullness analysis.

- It describes how a large class of existing data-flow analyses can be transformed to JIT analyses using a layered analysis approach.
- It shows how such a layered JIT analyzer can be built for taint analysis and applied to Android applications to find potentially insecure information flows.
- It empirically evaluates the implementation, focusing on performance and developer experience. Our experiments show that Cheetah returns results on real-world programs in less than a second, and developers can address leaks twice as fast compared to a batch-style analysis.

2 OVERVIEW

Despite years of work on eliminating false positives in static analysis tools, end-user experience tends to be overwhelming, even for the unsound (or optimistic) commercial tools; this is sometimes called the "wall of bugs" effect [22]. Observing how developers interact with static analysis tools, we highlight that: (1) reporting a warning is effectively useless if it is unlikely to be examined or result in a bug fix; and (2) some true warnings are abandoned because they are difficult to deal with [6]. To address both problems, batch-style tools typically draw developers' attention to specific high-priority warnings by applying post-analysis filtering and ranking to report the results *higher* or *lower* in the result list [12, 13]. Analyses that run in the background of the IDE can also report the results *earlier* or *later* in time, allowing developers to focus on a subset of warnings while the analysis computes further results. This approach of *interleaving* analysis and developer activities reduces the *perceived* analysis latency, improving the overall usability of an integrated analysis tool. Building on those observations, we define the following requirements for any sensible JIT analysis:

- **Prioritization:** The analysis must report the results most relevant to the user first.
- **Responsiveness:** To provide the users with immediate feedback on their changes, the analysis should report the earliest results quickly.
- **Monotonicity:** A reported issue cannot be refuted until the developer has fixed it: the analysis only *adds* warnings over time. Therefore, a JIT analysis is not a refinement of an imprecise pre-analysis.

2.1 Examples of JIT Analysis

Determining what is relevant to the user directly influences what should be reported first by a JIT analysis. We outline concrete examples for expressing various relevance metrics in three data-flow analyses.

2.1.1 Taint Analysis. A taint analysis tracks sensitive dataflows from sources to sinks to detect privacy leaks [16, 21, 39]. In Figure 1, a taint analysis reports two leaks: from the source on line 3 to the sinks on line 8 Ⓐ, and line 17 Ⓑ. The sink on line 12 is never reached, because line 11 overwrites the tainted variable x with non-sensitive data.

When writing code, developer attention is focused on the particular parts of the code that they are editing. Hence, it is sensible to prioritize warnings by *locality*, i.e., report those warnings that are closest to the user's working set first. For example, if the user is editing the main method in Figure 1, Ⓐ should be reported first, because it is located in the same method as the edit point. Ⓑ should be reported later, as it is located in a different class.

2.1.2 API Misuse Detection. To ensure correct API usage, analyses verify that programs follow a certain usage protocol [1]. In Figure 2, the analysis verifies that a cryptographic cipher is always initialized with init before a call to doWork. Result Ⓒ is harder to detect than Ⓓ, because the call to maybeInit on line 22 may resolve to either of the two implementations of the method. Applying an ordering by locality, a JIT analysis for API misuse detection can find Ⓓ before Ⓒ, because finding Ⓒ requires computing information over three different classes compared to two.

Another strategy based on *confidence* could prioritize monomorphic calls to polymorphic calls, as the latter are more likely to

Table 1: Layers of a JIT analysis for Android applications. L3 and L8 model the lifecycle of the event-based Android framework.

Layer	Name	The layer propagates the dataflows...
L1	Method	... in the same method as the current edit point. At method calls, the analysis stores the propagated information in memory to be resolved at later layers.
L2	Class	... along calls to methods in the same class as the current edit point.
L3	Class Lifecycle	... along lifecycle methods in the same class as the current edit point.
L4	File	... along calls to methods in the same file as the current edit point.
L5	Package	... along calls to methods in the same package as the current edit point.
L6	Project Monomorphic	... along the monomorphic calls in the project.
L7	Project Polymorphic	... along the polymorphic calls in the project.
L8	Android Lifecycle	... along lifecycle methods in the project, to handle interactions between the application components.

yield false positives. Similarly, a strategy based on *computational resources* could delay the computation polymorphic calls, as they create more data flows than monomorphic calls. In general, local results are not just the most relevant to the user's current task, they should also be computed precisely and quickly.

2.1.3 Nullness Analysis. A nullness analysis searches for null dereferences to avoid runtime errors. In Figure 3, a nullness analysis reports three warnings: Ⓔ because `f` points to `null`, Ⓕ because `f` and `g` must-alias after the assignment statement on line 42, and Ⓖ due to the may-alias on line 44. While Ⓔ takes minimal computation to find, Ⓕ and Ⓖ require additional alias information. In real-world programs, such flows can become exponentially more complex, and take minutes of computation to be reported, holding back the delivery of other simpler results that could be fixed in the meantime. With an ordering strategy by confidence, Ⓔ can be reported quickly, while alias information is computed to find Ⓕ and then Ⓖ.

Correcting the first warnings early enables the JIT analysis to update the results. For example, fixing Ⓔ, automatically fixes Ⓕ and Ⓖ as well. This early fix reduces the total number of warnings that are presented to the user.

3 JIT ANALYSIS THROUGH LAYERING

In this section, we discuss how one can transform an existing dataflow analysis into a locality-based JIT analysis. Our approach reorganizes the analysis into locality-based *layers*. The goal is to immediately report the results closest to the user's working set. Lower analysis layers run first, yielding the first results in a few seconds. The following layers enrich the analysis by computing increasingly complex results.

3.1 Locality-Based Layers for Android

To analyze Android applications, we propose a layered analysis that computes warnings by gradually increasing the analysis scope, i.e., by taking more code into consideration, starting at the current edit point. Table 1 shows the set of layers for this strategy. *Prioritization* comes by design, with a prioritization strategy based on locality. *Responsiveness* is ensured as lower layers require minimal class loading and computational resources. For example, only one class is loaded to compute results up to **L3**. *Monotonicity* is guaranteed by the internal implementation of each layer: if a layer cannot confirm a result, it delegates its computation to later layers.

Table 2: JIT analysis results for Figures 1–3, for the respective starting points: `main`, `encrypt`, and `main`.

	L1	L2	L3	L4	L5	L6	L7	L8
Taint	Ⓐ			Ⓑ				
API						Ⓓ	Ⓒ	
Null	Ⓔ Ⓕ Ⓖ							

3.2 Layered Analysis Examples

Table 2 shows the warnings that the JIT analyses described in Section 2.1 report using the layering system of Table 1 for the examples in Figures 1–3. The JIT taint analysis reports the direct leak Ⓐ at **L1**. Supposing that classes `A` and `B` are in the same file, Ⓑ is reported after the resolution of the call on line 7, at **L4**. The JIT API misuse detection reports Ⓒ and Ⓓ after the two calls to `maybeInit` on lines 22 and 26, respectively. Assuming that the calls are not in the same package as the `encrypt` method, Ⓒ is reported in **L7** and Ⓓ in **L6**. Since the layering system does not include alias-specific information, the three null dereferences Ⓔ, Ⓕ and Ⓖ are reported at **L1**.

3.3 Layering an Existing Analysis

To ease exposition, we describe how to transform into a layered JIT analysis such analyses that are distributive [35], i.e., in which flow functions f distribute over the merge operator: $f(x) \sqcap f(y) = f(x \sqcap y)$. Therefore, one can compute flows independently from one another and in any order, without loss of precision. Layering non-distributive analyses is possible but more complex, and we leave its description to future work.

3.3.1 Definitions. A *trigger* is a statement at which the JIT analysis pauses the propagation of certain data-flow facts to prioritize others. In Figure 1, the triggers are the two calls to `sendMessage`

Algorithm 1 Formalization of a JIT analysis

```
 1: procedure MAIN
 2:     $PQ := \{initialTask()\}$ //init priority queue
 3:     $computedTasks = \emptyset$
 4:     while $PQ \neq \emptyset$ do
 5:         pop task $t$ off priority queue $PQ$
 6:         if $t \notin computedTasks$ then
 7:             ANALYZE($t$)
 8:             $computedTasks \cup = \{t\}$
 9: procedure ANALYZE($\langle l, s_t, in \rangle$)
10:     $wl := \{s_t\}$ //init worklist
11:     $IN[s_t] = in$
12:     while $wl \neq \emptyset$ do
13:         pop $s$ off $wl$
14:         if $isTrigger(s)$ and $s_t \neq s$ then
15:             for $l' \in \{1..|layers|\}$ do
16:                 $in' := \{i \mid i \in IN[s], layer(s, i, l) = l'\}$
17:                 add new task $\langle l', s, in' \rangle$ to $PQ$
18:         else
19:             $OLD := OUT[s]$
20:             $IN[s] := \sqcup\{OUT[p] \mid p \in preds(s)\}$
21:             $OUT[s] := f_s(IN[s])$
22:             if $OLD \neq OUT[s]$ then
23:                 $wl \cup = succ(s)$
```

PQ returns tasks with the lowest priority layers first. $initialTask()$, $isTrigger()$, $layer()$, $f_s()$ are parameters of the analysis.

on lines 6 and 7. At those triggers, the JIT analysis propagates `u` before propagating `s` and `t` to prioritize reporting Ⓐ, because it is in the same method as the starting point `main`. The choice of propagating `s` or `t` next depends on the *priority layers*. Data-flow facts created at a trigger create a *task*. In Figure 1, two tasks are created: one with the initial set `{s}` with priority **L2**, because the call to `sendMessage` on line 6 resolves to a call in the same class, and one with the set `{t}` with priority **L4**, because the call to `sendMessage` on line 7 resolves to a call in the same file. The analysis executes the first task because its priority layer is lower, propagating `s` until the next trigger, or the end of the program, and then executes the second task to report Ⓑ.

3.3.2 *Algorithm.*

We present Algorithm 1 as a general recipe to transform a distributive data-flow analysis in a JIT analysis.

The procedure *analyze*, excluding lines 14–17, represents a standard fixed-point iteration for a traditional data-flow analysis that applies the flow function f_s to the statements of a program (line 21) until the *OUT* sets remain unchanged. The transformation to a JIT analysis divides this large fixed-point iteration into smaller ones (tasks). At trigger points, the JIT analysis forces an intermediate fixed-point by not modifying the *OUT* set (line 14), stopping the current analysis task prematurely. Non-trigger statements are handled in the same way that the traditional analysis does (lines 18-23).

To eventually compute the same results as the traditional analysis, the JIT analysis creates new tasks at triggers, and adds them to the priority queue *PQ* to be executed later (lines 15-17). When a task is executed, the JIT analysis pops the next highest-priority task from *PQ*. It then creates a new instance of the traditional analysis, and initializes it with the appropriate *IN* set, to continue the propagation where the previous task stopped. The role of the priority queue is to prioritize task propagation to report certain warnings first. This is determined by *layer(s,i,l)*, returning the priority layer *l'* of the new task that will continue propagating the fact *i* at statement *s*, knowing that it was paused at layer *l*.

While there are multiple ways of instantiating the JIT concept, Algorithm 1 requires minimal changes to adapt existing analyses: (1) a priority queue is added to the solver, (2) no changes are introduced into the original flow functions $f_s()$, leaving the definition of the data-flow problem entirely unmodified, and (3) the analysis writer can instantiate different priority systems independently from the solver and the flow functions through $initialTask()$, $isTrigger()$, and $layer()$.

Termination. Algorithm 1 extends an existing traditional analysis. If the traditional analysis terminates, the inner loop (line 12) is guaranteed to terminate for all analysis instances, because the algorithm does not modify the *IN* and *OUT* sets. The outer loop (line 4) also terminates, because the number of created tasks is bounded. Tasks depend on their associated set of facts. If the data-flow lattice of the traditional analysis is bounded, the number of facts, therefore tasks, is also bounded. Line 6 checks that no task is computed twice, ensuring termination and improving efficiency.

Soundness. To be as sound as the base traditional analysis, the JIT analysis checks that every data-flow fact created by the flow functions of the traditional analysis is assigned to at least one layer. Algorithm 1 partitions the *IN* set of a statement into smaller sets (line 16). For this operation to be safe, the data-flow facts should be separable, i.e., the analysis problem should be distributive so that data-flow facts can be independently distributed between the layers. We further improve efficiency by assigning each data-flow fact of an *IN* set to exactly one layer.

Requirements. We summarize the requirements for creating a JIT analysis according to Algorithm 1:

- The base analysis must terminate.
- The analysis problem must be distributive.
- The priority layers must provide a complete and disjoint partitioning of the *IN* set.

Layering by locality as described in Table 1 fulfills these requirements by using method calls as triggers, and partitioning *IN* sets according to their callees. Other layering strategies can be used to fit other problems, e.g., by confidence or computational resources.

4 CHEETAH: A JIT TAINT ANALYSIS

Following our proposed layered approach, we have implemented Cheetah, a JIT taint analysis that detects data leaks in Android applications. Cheetah is built on top of the Soot analysis framework [40] and the Heros IFDS solver [7]. IFDS is a framework for solving inter-procedural finite distributive subset problems as graph-reachability problems on a directed graph representing the facts of interest to the analysis at each program point within the *exploded super-graph* [35]. We use IFDS as a succinct way to define Cheetah based on a simple IFDS taint analysis that tracks explicit data-flows. Cheetah uses the sources and sinks definitions

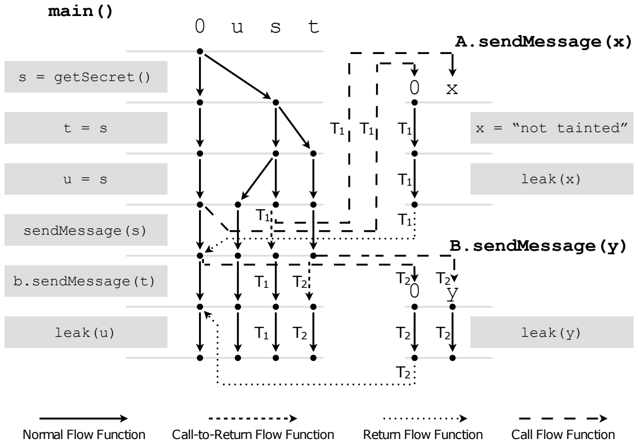


Figure 4: IFDS propagation for the example in Figure 1. Edges are labeled by the analysis task that creates them. The unlabeled edges are created by task T_0.

described by Rasthofer et al. [33]. We have made Cheetah publicly available online, along with all experimental data [10].

4.1 IFDS Flow Functions

The IFDS flow functions map each existing data-flow fact to its successors. For a taint analysis, a typical flow function (1) generates new taint flows if it encounters source methods; (2) kills taints if the tainted variable is overwritten by non-tainted data; or (3) propagates taint if tainted references are assigned to other references. There are four types of IFDS flow functions, which we illustrate in Figure 4 for the running example from Figure 1.

- **Normal flow functions**: are applied at each statement that is not a call, e.g., s is mapped to t on line 4.
- **Call flow functions**: are applied at each call statement. They propagate the data-flow facts from the caller to the callee, e.g., t is mapped to y on line 7.
- **Return flow functions**: are applied at call statements, and map the facts from the callee back to the caller.
- **Call-to-return flow functions**: are applied at each call statement, and propagate the facts that are not affected by the call, e.g., u is mapped to u on line 7.

For a detailed presentation of Cheetah's flow functions, we refer the reader to our technical report [30].

4.2 Layered Taint Analysis

Using the layers in Table 1 and Algorithm 1 to transform a traditional IFDS taint analysis into Cheetah, we define:

$$initialTask() = \{L1, startPoint, \{0\}\}$$
$$isTrigger(s) = s.containsMethodCall()$$
$$layer(s, i, l) = distance(s.callee, startPoint)$$

Cheetah marks all call sites as triggers, meaning that the data-flow propagations at call sites are paused and continued in subsequent tasks. The layer that is assigned to a fact at a call site is determined by the distance (in terms of the priority layers) between the callee and the start point of the analysis, which is the method containing the current edit point. For example, if Cheetah encounters a call to a method that is in the same file but not the same class as the starting point, the new task is assigned **L4**. As a result, one task creates as many tasks as the number of call sites it contains. New tasks are added to the priority queue PQ, and executed in order of distance from the starting point. To adapt Algorithm 1 to the IFDS framework, we apply the following changes:

- Every time a task is executed (line 7), Cheetah creates a new IFDS instance starting at the task's start statement (s_t), and initializes it with the facts contained in its *in*-set. To reuse previously computed results, the state of the IFDS solver is carried over from one instance to the next.
- The priority queue PQ is initialized with the task $\{L1, stmt, \{0\}\}$, where $stmt$ is the first statement of the currently edited method, and 0 is the initial fact for a standard IFDS propagation.
- To create new tasks at call sites, Cheetah slightly modifies the call flow function of the traditional analysis to stop the propagation of data-flow facts at call sites (by returning the empty set), except when the call is the start statement of the current task. At a call site, Cheetah collects the variables that need to be propagated further (i.e., the parameters of the call, the static variables, and the receiver of the call) in an $inSet$. A new task $\{layer(stmt), stmt, inSet\}$ is then added to PQ to be executed later. This change corresponds to lines 14–17 in Algorithm 1.
- The normal, return, and call-to-return flow functions remain the same as the traditional analysis.

Example: Applying Cheetah to the example in Figure 1 results in the following steps (also shown in Figure 4):

(1) The user triggers the analysis at the `main` method. Task T_0 = $\{L1, line\ 3, \{0\}\}$ is enqueued.
(2) T_0 is executed, resulting in the unlabeled edges.
  (a) Ⓐ is found and reported
  (b) Task $T_1 = \{L2, line\ 6, \{0, s\}\}$ is created.
  (c) Task $T_2 = \{L4, line\ 7, \{0, t\}\}$ is created.
(3) T_1 is executed, resulting in the edges labeled T_1.
(4) T_2 is executed, resulting in the edges labeled T_2, and Ⓑ is reported.

5 EMPIRICAL EVALUATION

We empirically evaluate Cheetah by comparing it to Base, the traditional IFDS taint analysis that we transformed to obtain Cheetah. Our experiments address the following research questions:

RQ1: How *responsive* is Cheetah compared to Base?
RQ2: How *early* does Cheetah report warnings?
RQ3: Are the initial findings of Cheetah easier to *interpret* than later ones?

5.1 Tools

Cheetah and Base have the same flow functions, except for the slight modification to the call flow function described in Section 4.2 which enables us to make Base just-in-time. To model the Android lifecycle, Base uses a dummy `main` method that models the Android lifecycle of an application [2]. Cheetah models the Android lifecycle on a per-class basis by distributing the dummy `main` over

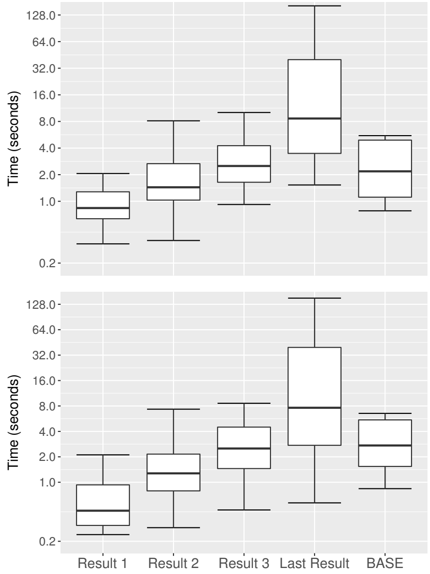


Figure 5: Time to report results (in log scale) for CHEETAH and BASE, starting at SPB (top) and SPS (bottom).

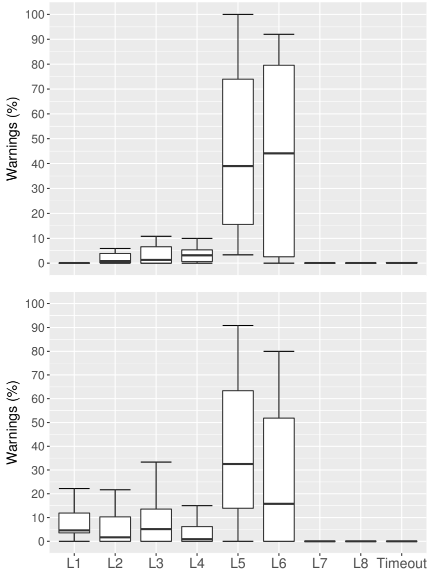


Figure 6: Percentage of warnings reported at each layer in CHEETAH with SPB (top) and SPS (bottom).

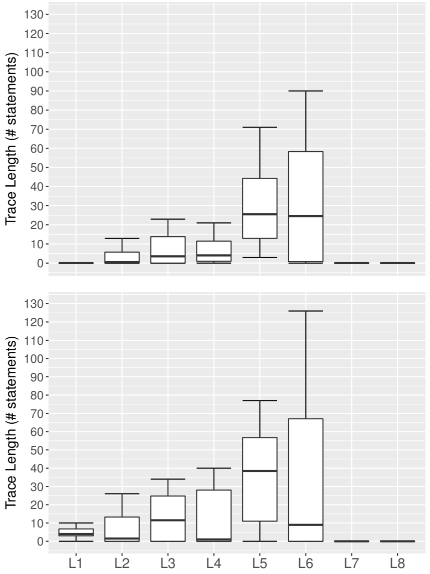


Figure 7: Trace length of the warnings reported at each layer in CHEETAH with SPB (top) and SPS (bottom).

the layering system. Currently, both Base and Cheetah resolve virtual calls by using a simple CHA-based callgraph [5].

5.2 Experimental Setup

Our benchmark suite contains 14 Android applications selected from F-Droid [17], such that each application has a GitHub repository with more than one commit and is available for mining in Boa [15]. We used Boa to collect the methods modified in each commit, referred to as SPB (Starting Points Boa). Each application has at least 26 unique SPB (min: 26, max: 316, median: 127). We ran two experiments. In the first one, we ran Cheetah 20 times for each application using randomly selected SPB as starting points for the analysis. In the second experiment, we chose the sources of known data leaks that were obtained from the first experiment as the starting points, referred to as SPS (Starting Points Sources). SPS represent cases when the user is investigating a particular bug, SPB, cases when the user does not actively use Cheetah during code development. Base has one starting point: a dummy main method that acts as the entry point to the Android application [2].

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, and limited the Java heap space to 1 GB.

5.3 Results

RQ1: How *responsive* is CHEETAH compared to BASE? We have measured the time that Cheetah takes to report the first, second, third, and last result when it starts at SPB and at SPS. We compare those times to the time it takes Base to report its final results. Figure 5 shows, in log scale, the *total response time* for those quantities, which includes the *overhead time* taken by Cheetah to load and process the initial set of classes. Across our benchmark, Cheetah reports the first result in a median time of less than 1 second when it starts at SPB and a median of less than 0.5 seconds when it starts at SPS. These results are below Nielsen's 1 second recommended threshold for interactive user interfaces, suggesting that Cheetah usually allows the "user's flow of thought to stay uninterrupted" [31]. Cheetah reports its last result in a median time of 9.03s and 7.79s when it starts at SPB and SPS, respectively. Both medians are larger than the median times of 1.85s (SPB) and 2.13s (SPS) that Base takes to report its final results. This is because Cheetah analyzes parts of the program that are not reachable from its main entry points, a feature that traditional analyses such as Base do not offer. Any analysis imprecision in those parts propagates to the other computations, making the analysis perform more work than strictly necessary. Nevertheless, such a feature is desirable for real-life code development scenarios, which we discuss in Section 8.

Cheetah returns the first result in less than one second, allowing the developer to remain focused.

RQ2: How *early* does CHEETAH report warnings? One of the main goals of Cheetah is to better help software developers detect bugs located around their working sets compared to using traditional analyses. This means that Cheetah should ideally report most of its warnings in earlier layers. Figure 6 shows that, across our benchmark, when Cheetah starts at SPB, a median of 38.97% of the warnings is reported in **L5** and a median of 44.12% in **L6**. Starting at SPS, Cheetah reports a median of 32.56% warnings in **L5** and a median of 15.77% in **L6**. With SPS, Cheetah reports more warnings in earlier layers: a median of 4.58% in **L1** and a median of 5.13% in **L3**. Unlike SPB, SPS simulates scenarios where users are interested in the analysis results. In those cases, 33.3% of the warnings are reported at **L1**-**L4** on average, against 11.6% for

SPB. This shows that if Cheetah is guided towards the points of interest in a program, more warnings are reported at earlier layers. Additionally, in Figure 5, we see that the first results are returned faster with SPS (e.g. medians of 1 second (SPB) and 0.5 seconds (SPS) for the first result). Therefore, starting at SPS is optimal when the user requires analysis updates while fixing a particular warning.

After Cheetah reports a leak, a separate module retrieves the paths between the leak's source and sink to provide the user with more information. The process of retrieving those paths times out in less than 1% of all the cases (average: 0.81%, median: 0%). It is important to note that, for those timeouts, Cheetah itself does not time out, but the path-finding module does.

Across our benchmark, no results are reported in **L7**, because none of the applications pass sensitive information through polymorphic calls. Similarly, no warnings are reported in **L8**, because Cheetah currently does not support inter-component flows.

> Cheetah reports most of the warnings in **L5**-**L6**. If directed to known sources of bugs, Cheetah reports the first warnings faster, and it reports more warnings in earlier layers.

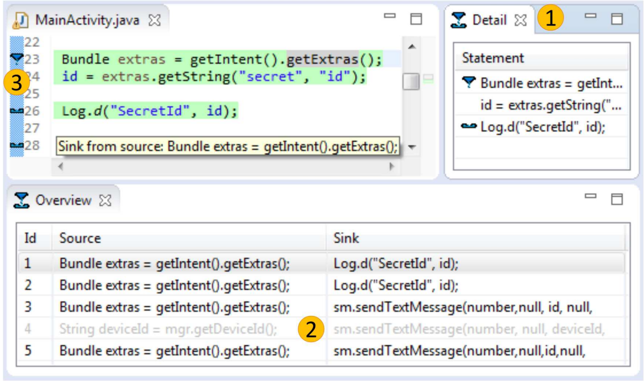
RQ3: Are the initial findings of CHEETAH easier to *interpret* than later ones? The quick response time of Cheetah is only useful if the first few warnings that it reports are easy to *interpret* by the user. Otherwise, the user will spend most of her time trying to trace her way through the program to fix that warning. We approximate the ease of interpretation of the initial warnings that Cheetah reports by computing the *trace length*: the number of statements between the source and the sink for a given warning. Figure 7 shows the trace lengths for the warnings that appear in each layer of Cheetah. When Cheetah starts at SPB, the median length of the traces for the initial layers **L1**-**L4** is 0, 1, 4, and 4 statements, respectively. For later layers, Cheetah reports more complex warnings with longer trace lengths: medians of 26 and 25 statements for layers **L5** and **L6**, respectively. Starting at SPS, Cheetah reports more warnings in earlier layers. In such a case, the median length of the traces that Cheetah reports for the initial layers **L1**–**L4** is a median of 4, 2, 12, and 1 statements, respectively.

> In layers **L1**–**L4**, Cheetah reports warnings with shorter traces than later layers, making them easier to interpret.

6 GRAPHICAL USER-INTERFACE

Interleaving analysis and developer activities requires careful reporting. Otherwise, warnings can literally become moving targets in result lists as new ones are found and others are fixed, which confuses the developer. Figure 8 captures the main GUI elements of Cheetah that we have introduced to support reporting warnings over time. These features, described below and highlighted with the corresponding numbers in the figure, address the observations made during our pilot study. A demo of the GUI is available online [10].

1. **Views:** To avoid overwhelming the users by showing them all warnings and their details in one place, we separate the information into two views: *Overview* lists all reported warnings, and *Detail* presents the path of a selected leak, offloading the amount of information contained in *Overview*.



```
Bundle extras = getIntent().getExtras();
id = extras.getString("secret", "id");

Log.d("SecretId", id);
```

Sink from source: Bundle extras = getIntent().getExtras();

Statement
Bundle extras = getInt...
id = extras.getString("...
Log.d("SecretId", id);

Id	Source	Sink
1	Bundle extras = getIntent().getExtras();	Log.d("SecretId", id);
2	Bundle extras = getIntent().getExtras();	Log.d("SecretId", id);
3	Bundle extras = getIntent().getExtras();	sm.sendTextMessage(number,null, id, null,
4	String deviceId = mgr.getDeviceId();	sm.sendTextMessage(number, null, deviceId,
5	Bundle extras = getIntent().getExtras();	sm.sendTextMessage(number,null,id,null,

Figure 8: GUI elements of Cheetah.

2. **Color-coded warnings:** Warnings in Cheetah have three states: *active* (confirmed by the latest analysis run), *computing* (found by the previous analysis run, but not yet confirmed by the current run), and *fixed.* Cheetah displays active warnings in black and computing warnings in gray. Fixed warnings are grayed out for one run of the analysis, and removed from the view at the next run. This feature provides a light history of fixed leaks, allowing users to quickly check if a fix was effective.
3. **Descriptive icons:** Cheetah displays source and sink icons on the left gutter. When a warning is grayed out in the Overview, the corresponding icons are also grayed out. Tooltips provide additional information on the leaks.
4. **Seamless run:** To integrate Cheetah in Eclipse, we trigger the analysis every time the project is built. Cheetah hooks into Eclipse's incremental builder and re-runs the analysis starting from the method that has the focus.
5. **Other features:** Cheetah also provides a few features that are not JIT-specific, such as highlighting the path of the leak that the user is currently examining.

7 USER STUDY

In a user study, we evaluate how a JIT analysis integrates into the development workflow compared to a batch-style analysis. 18 participants performed development tasks using Cheetah or Base to keep the number of data leaks to a minimum.

An earlier pilot study with 11 participants showed that the prototype had poor usability. GUI-related issues prevented participants from focusing on their tasks. We fixed those issues before conducting the study we describe in this section.

7.1 Setup

7.1.1 Participants. Our study includes 18 participants of varying backgrounds (9 academics and 9 professional developers), with different skill levels in terms of Android development and knowledge of taint tracking static analysis tools. In the following, we identify them as **P1** ... **P18**[1].

[1] We discard **P17**'s data, as (s)he encountered a user interface bug and was unable to properly perform the tasks.

7.1.2 Tasks. In order to evaluate the influence of the analysis tools on the development workflow, each participant performed a development task: removing code duplicates in an Android application. At the same time, we asked them to keep the number of *data leaks* to a minimum. To help detect potential data leaks, Cheetah or Base were provided as Eclipse plugins. To fix the leaks, we provided the participants with data sanitization APIs. Each task was limited to 10 minutes.

7.1.3 Protocol. Each participant performed one task per tool (Base and Cheetah). Before each task, the participants warmed up on a small Android application to get used to the tool. The order of the tools was randomized, so that half of the participants started with Cheetah, and the other half with Base. Afterwards, the participants filled a comparative questionnaire and were interviewed in person. The questionnaire, responses, and the interview protocol are available online [10].

7.1.4 Test Applications. The warm-ups were performed on a small, artificial Android application that contained 6 simple data leaks. The two tasks were performed on a real-life application from F-Droid [17]: Bites, a basic cookbook app. Due to the limited time (10 minutes per task), we have modified Bites to add data leaks around code duplications, resulting in a total of 106 more complex data leaks. This ensured that participants encountered data leak warnings while conducting their duplication removal task. In the pilot study, some participants had spent most of their time handling code duplicates not related to any data leaks.

7.1.5 GUI. To reduce any GUI-induced bias, the GUI used for Base is almost identical to Cheetah's. For this study, Base emulates a batch-style tool: Base is not triggered on code build, but by pressing a button. A popup then blocks the GUI to prevent the user from modifying the code while the analysis is running. All results are shown at the same time after the analysis finishes.

7.2 Results

Figure 9 shows, in log scale, the distribution of the time taken to fix a leak for the two main tasks. The reported numbers take into account the time taken to fix a leak, discarding the time needed by the participant to understand the code, the tool, or to discuss the solution before implementing it. For Tasks 1 and 2, participants using Cheetah took half as long as the participants using Base to fix a leak (0.53× and 0.45×, respectively). For Task 1, the median time to fix a leak using Base was 63 seconds, compared to 33.5 seconds per leak for Cheetah users. The times reported for Task 2 are lower: 54.5 seconds per leak for Base and 24.5 seconds per leak for Cheetah. This is because by Task 2, participants are more familiar with the application and the tasks. We also note that across both tasks, the participants were significantly faster when using Cheetah compared to Base ($p < .01$, Wilcoxon Rank-Sum test), regardless of whether they used Base first and Cheetah second (2.6× faster), or Cheetah first and Base second (1.6× faster).

We also observed that in the 10 minutes allocated to each task, Cheetah users fixed more leaks than Base users, with a median of 2 leaks for Base users and 4 leaks for Cheetah users in Task 1. For Task 2, the medians are 3 leaks for Base users and 4 leaks for Cheetah users. The Wilcoxon Rank-Sum test failed to detect a significant difference in the number of leaks fixed ($p = .31$).

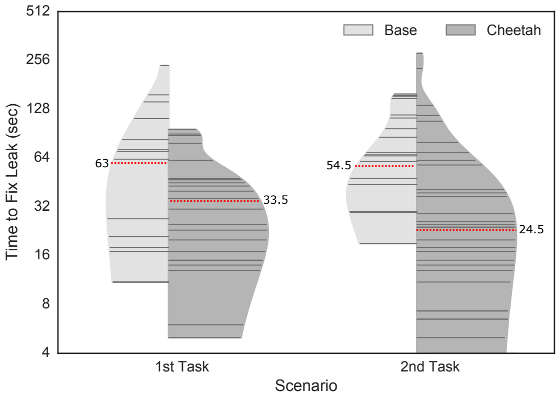


Figure 9: Violin plot representing the distribution of the times to fix leaks across all participants, by task and tool used during the task. Each horizontal line represents data leaks fixed in the corresponding time. The length of a line represents the number of leaks fixed in that time. Dashed lines are medians.

Using Cheetah enables users to fix leaks twice as fast compared to using Base.

7.3 Questionnaire

After the four tasks, the participants filled a questionnaire comprised of 29 questions designed to assess the merits of the two approaches, also providing some open-ended comments to compare Cheetah to Base. The participants answered several 5-point Likert-type questions from the System Usability Scale (SUS) [8], a questionnaire designed to measure the effectiveness and efficiency of a system, and rated both tools using a Net Promoter Score (NPS) [34], an 11-point Likert scale that measures their likelihood of recommending the tool to a friend.

Overall, the participants responded positively to Cheetah. Among the participants who rated Cheetah and Base on the NPS question ($n = 16$), Cheetah's mean score is 7.4 (out of 10) compared to a mean score of 2.7 for Base. According to the aggregated SUS scores, 12 participants rated Cheetah higher than Base. Using a Wilcoxon Signed-Rank test [41], we observed significant ($p < 0.05$) differences between these aggregated scores and participant's responses on 4 of the individual SUS questions. Compared to Base, participants less likely found Cheetah unnecessarily complex or cumbersome (-0.6 mean response each). Moreover, the participants responded that they were more likely to use Cheetah frequently (+0.7 mean response), and more likely found its functions well-integrated (+0.5 mean response).

7.4 Interviews

During the individual interviews, the participants were asked to detail their experience of the tools, focusing on the percieved differences, in particular waiting times, integration of the tools in the IDE, and warning ordering. The interviews lasted 14 minutes on average (min: 10 minutes, max: 23 minutes). We now present notable comments and behavior.

Quick Updates. In total, 12 participants found Cheetah's quick updates useful, noting this feature as the main advantage of the tool. Professional developers in particular, noted that this system is "much more comfortable, and what I would expect in the Eclipse environment" (**P7**). **P2** noted that Cheetah "integrates well into the Eclipse build-on-save paradigm". This results in a "more fluent workflow" (**P9**), as opposed to Base, which proved more interruptive to the participants: "having to wait interrupts the coding and thinking process" (**P6**). **P4** explained from their personal experience with UI-blocking compilation tools that they "do a context switch in your head. [...] When you are back to the actual work, you might have forgotten what you wanted to do". In summary, participants felt that for code development, Cheetah was less interruptive, as it allowed them to deviate less from their coding tasks.

Ordering. **P8**, a professional Android developer, noticed the order in which the results were reported with Cheetah, and commented, when using Base: "When I'm in one class, I get familiar with it, and when I click on a warning, it takes me to a completely different class, and I have to get used to it again", which further validates the choice of layers in Cheetah. **P18**, in particular, handled the leaks in the presented order. **P18** fixed all encountered leaks when using Cheetah, but skipped most of the first warnings when using Base after deeming their traces "too long". We see that reporting warnings following Cheetah's layers positively affects participant performance and integrates more discretely in their workflow.

Performance. Two expert participants expressed performance concerns about Cheetah running too often on big projects: "if the analysis affects the performance, I would like to have a button to control it" (**P13**).

7.4.1 Comparison. The participants were asked in which cases they would use one tool rather than the other.

- Twelve participants reported Cheetah is best suited for code development. **P9**, in particular, noted that it would make the development task slightly harder, but it would "force me to write better code from scratch".
- Two participants were concerned about using Cheetah for big projects because "if it has a big impact on the CPU, it might be annoying and I might not be as productive" (**P4**).
- Eleven participants noted that Base should be used infrequently or in situations where debugging and coding are separated: "after a milestone" (**P9**), "creating reports for software" (**P7**).
- No participants reported they would use Base for code development.

Twelve out of 17 participants preferred Cheetah for code development due to its quick updates. Two participants expressed concerns about CPU overhead. Cheetah's inherent ordering helps code developers perform bug-fixing tasks.

8 DISCUSSION

A traditional whole-program analysis usually starts from a main method and propagates through the code that is reachable from there. This approach is ill-suited for the scenario of code development, where developers often work on new features in incomplete programs that may not even have a main method. Cheetah provides full code coverage by artificially creating tasks that are not naturally induced by the code base. In addition to the tasks that it naturally creates, a task on **Lx** artificially creates all tasks on **Lx-1** which are in its scope and a task on **Lx+1** initialized with its own starting point. Therefore, the first task of the analysis, which is at **L1**, generates a task at **L2**, which in turn generates a task at **L3**, eventually generating a task at **L8**. This last task encompasses the whole project, therefore, the set of tasks at **L7** that it creates also covers the whole project, eventually resulting in a full coverage of the source code at lower layers. An extra check ensures that no task is executed twice. Because of the priority queue, tasks with lower layers are executed first. Therefore, the tasks of **L1** generated by a task of **L2** will be executed before the task of **L3** generated by this same **L2** task is executed. This ensures Cheetah's prioritization by locality. This approach ensures that the entire code base is analyzed and enables Cheetah to help software developers reason about unreachable code, a property that a traditional IFDS-based taint analysis does not provide.

In our evaluation, the IFDS taint analysis Base uses a dummy `main` as described by Arzt et al. [2] and only reports leaks in those parts of the code that are reachable from that method using a CHA call-graph. On the other hand, Cheetah provides full code coverage, resulting in more reported warnings and a longer overall analysis time. In our empirical experiments, Cheetah reports 2.1× more warnings compared to Base (min: 1×, max: 10×, geometric mean: 2.06×). Since Base and Cheetah have the same IFDS flow functions, they have the same soundness and precision by construction. By covering more of the code base, Cheetah provides the code developer with a more relevant result set than traditional analyses such as Base.

A JIT analysis's layering system can support more complex layering schemes. For example, in Cheetah, more layers can be added to improve the aliasing strategy. Currently, Cheetah computes aliases intra-procedurally, and ignores inter-procedural aliases. A more complex aliasing strategy could be instantiated by computing aliases together with the taint analysis such that both analyses share the same scope. When the analysis scope widens, more aliases are discovered, adding more taints to the analysis. A whole-program alias analysis could also be used in one of the later layers operating on the project scope.

9 RELATED WORK

Given the vast amount of research on static analysis, we focus this section quite narrowly, highlighting the interactions between static analysis tools and developers.

9.1 Human Aspects of Static Analysis Tools

Several researchers have studied the usage of static analysis tools by developers. Sadowski et al. [36] report that most Google developers find static analysis warnings usable. Phang et al. [23] found that a program flow visualization tool helps developers quickly triage warnings. Ayewah and Pugh [3] found that checklists helps developers consistently evaluate warnings. In an experiment, Smith et al. [38] characterized the information needs of developers while addressing warnings. In contrast, our work focuses on smoothly integrating static analysis warnings into developers' workflows.

Several human studies have highlighted challenges related to workflow integration. Johnson et al. [22] recorded interviewees stressing the importance of integrating static analysis into their workflows. Lewis et al. [26] found that almost all interviewed developers agreed that static analysis should not disrupt their workflow. Through interviews and surveys, Xiao et al. [43] and Witschey et al. [42] found that developers whose security tools help them do their work quickly report being more likely to adopt those tools. Christakis and Bird [11] interviewed and surveyed Microsoft developers who complained that existing tools are too slow and do not fit into their workflow. Accordingly, our work aims to address workflow integration problems by providing relevant static analysis results quickly.

9.2 Warning Prioritization

Researchers have proposed several ways to prioritize which warnings developers should address first. Industrial tools tend to use heuristics, such as FindBugs [18], which classifies warnings as low, medium, or high priority. Surveying the research, Muske and Serebrenik [28] organize prioritization approaches into three main categories: statistical, historical, and user-feedback. As an example of a user-feedback based approach, Heckman and Williams [20] use machine learning to prioritize *actionable* warnings over *unactionable* ones. As an example of a history-aware approach, Kim and Ernst [24] use code history to prioritize defects. Other approaches do not easily fit into these categories. For example, Shen et al. [37] deprioritize predicted false positives, then use developer feedback for future prioritization. As another example, Liang et al. [27] use resource leak defect patterns to prioritize potential resource leaks. While prior approaches prioritize using the warning or the code, our approach instead (1) prioritizes using a developer's working context, and (2) uses that context to guide the analysis itself.

9.3 Result Presentation

Several prior researchers have investigated how to best present analysis results to the user. For example, Solstice [29] focuses on non-disruptive reporting. It runs an offline analysis on a replica of the developer's workspace, reporting results in a non-disruptive manner. In contrast, Cheetah is an interactive analysis of the original codebase in the IDE, without code replication.

Parfait [12, 13] runs an initial bug detector then cascades different analyses in layers of an increasing order of complexity and a decreasing order of efficiency to confirm the initial findings. Unlike Cheetah, one layer in Parfait may invalidate some of the bugs that a previous layer has already reported. In Cheetah, each layer uses previously computed information to detect *new* bugs and does not invalidate previously reported warnings, minimizing the disruption in the developer's workflow.

Cheetah reports warnings in a similar way to how Eclipse's incremental compiler reports errors to a user while editing source files [25]. This is the same approach used by ASIDE [44], an Eclipse plugin that detects security vulnerabilities in Java programs. ASIDE incrementally reports errors to the user by only analyzing recent code changes. Cheetah is a whole-program analysis, but it still incrementally reports warnings to the user by starting at specific points of interest in the program (e.g., a recently modified method). Although incremental analyses compute minimal changesets and Cheetah recomputes everything at each run, Cheetah consistently provides the first results quickly while some changesets can cause an incremental analysis to fully recompute its results. Nevertheless, we plan to incrementalizeCheetah to improve the responsiveness of its later layers **L5**–**L8**.

10 CONCLUSION

We have presented the novel concept of JIT analysis that interleaves the processes of code development, static analysis execution, and bug fixing, through a layered static analysis approach. We have shown how to obtain a JIT analysis by modifying a base distributive data-flow analysis with minimal changes. We have also provided Cheetah, an implementation of a JIT taint analysis for finding privacy leaks in Android applications, and evaluated it on real-world applications. Our empirical results, questionnaire results, and a detailed user study show that Cheetah's quick updates and ordering strategy make it particularly well-suited for integrating bug fixing within the natural flow of code development.

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] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. 2015. Towards secure integration of cryptographic software. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 1–13.

[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 *Programming Language Design and Implementation (PLDI)*. 259–269. DOI: http://dx.doi.org/10.1145/2594291.2594299

[3] Nathaniel Ayewah and William Pugh. 2009. Using Checklists to Review Static Analysis Warnings. In *International Workshop on Defects in Large Software Systems (DEFECTS)*. 11–15. DOI: http://dx.doi.org/10.1145/1555860.1555864

[4] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *International Symposium on Software Testing and Analysis (ISSTA)*. 241–252. DOI: http://dx.doi.org/10.1145/1831708.1831738

[5] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 324–341. DOI: http://dx.doi.org/10.1145/236337.236371

[6] 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. *Communications of the ACM* 53, 2 (2010), 66–75. DOI: http://dx.doi.org/10.1145/1646353.1646374

[7] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *International Workshop on State of the Art in Java Program Analysis (SOAP)*. 3–8. DOI: http://dx.doi.org/10.1145/2259051.2259052

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

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

[10] Cheetah. 2017. https://blogs.uni-paderborn.de/sse/tools/cheetah-just-in-time-analysis/. (2017).

[11] Maria Christakis and Christian Bird. 2016 (to appear). What Developers Want and Need from Program Analysis: An Empirical Study. (2016 (to appear)).

[12] Cristina Cifuentes. 2008. Parfait - A Scalable Bug Checker for C Code. In *Source Code Analysis and Manipulation (SCAM)*. 263–264. DOI:http://dx.doi.org/10.1109/SCAM.2008.21

[13] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. 2012. Transitioning Parfait into a Development Tool. *IEEE Security & Privacy* 10, 3 (2012), 16–23. DOI:http://dx.doi.org/10.1109/MSP.2012.30

[14] Coverity. 2017. http://www.coverity.com/. (2017).

[15] 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 *International Conference on Software Engineering (ICSE)*. 422–431.

[16] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Operating Systems Design and Implementation (OSDI)*. 393–407.

[17] F-Droid. 2017. Free and Open Source Android App Repository. https://f-droid.org. (2017).

[18] FindBugs. 2017. http://findbugs.sourceforge.net. (2017).

[19] HP Fortify. 2017. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/. (2017).

[20] Sarah Heckman and Laurie Williams. 2009. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing, Verification and Validation (ICST)*. 161–170.

[21] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 106–117. DOI:http://dx.doi.org/10.1145/2771783.2771803

[22] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering (ICSE)*. 672–681. http://dl.acm.org/citation.cfm?id=2486877

[23] Yit Phang Khoo, Jeffrey S Foster, Michael Hicks, and Vibha Sazawal. 2008. Path projection for user-centered static analysis tools. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 57–63.

[24] Sunghun Kim and Michael D Ernst. 2007. Which warnings should I fix first?. In *Foundations of Software Engineering (FSE)*. 45–54.

[25] Andrew Jensen Ko and Brad A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Conference on Human Factors in Computing Systems (CHI)*. 387–396.

[26] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. Does bug prediction support human developers? Findings from a Google case study. In *International Conference on Software Engineering (ICSE)*. 372–381. http://dl.acm.org/citation.cfm?id=2486838

[27] Guangtai Liang, Qian Wu, Qianxiang Wang, and Hong Mei. 2012. An effective defect detection and warning prioritization approach for resource leaks. In *Computer Software and Applications Conference (COMPSAC)*. 119–128.

[28] Tukaram Muske and Alexander Serebrenik. 2016. Survey of Approaches for Handling Static Analysis Alarms. In *Source Code Analysis and Manipulation (SCAM)*. http://www.win.tue.nl/~aserebre/SCAM2016.pdf

[29] Kivanc Muslu, Yuriy Brun, Michael D. Ernst, and David Notkin. 2013. Making Offline Analyses Continuous. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 323–333. DOI:http://dx.doi.org/10.1145/2491411.2491460

[30] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2016. *Just-in-Time Static Analysis*. Technical Report doi:10.7939/DVN/10859. University of Alberta.

[31] Jakob Nielsen. 1994. *Usability Engineering*. Elsevier.

[32] PREfast. 2017. https://msdn.microsoft.com/en-us/library/ms933794.aspx. (2017).

[33] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Network and Distributed System Security Symposium (NDSS)*.

[34] Frederick F Reichheld. 2003. The one number you need to grow. *Harvard Business Review* 81, 12 (2003), 46–55.

[35] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages (POPL)*. 49–61. DOI:http://dx.doi.org/10.1145/199448.199462

[36] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*. 598–608.

[37] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. Efindbugs: Effective error ranking for findbugs. In *International Conference on Software Testing, Verification and Validation (ICST)*. 299–308.

[38] Justin Smith, Brittany Johnson, Emerson R. Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Foundations of Software Engineering (FSE)*. 248–259.

[39] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Programming Language Design and Implementation (PLDI)*. 87–97.

[40] 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 *Compiler Construction (CC)*. 18–34. DOI:http://dx.doi.org/10.1007/3-540-46423-9_2

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

[42] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *Foundations of Software Engineering (FSE)*. 260–271. DOI:http://dx.doi.org/10.1145/2786805.2786816

[43] Shundan Xiao, Jim Witschey, and Emerson R. Murphy-Hill. 2014. Social influences on secure development tool adoption: why security tools spread. In *Computer Supported Cooperative Work & Social Computing (CSCW)*. 1095–1106. DOI:http://dx.doi.org/10.1145/2531602.2531722

[44] Jing Xie, Heather Lipford, and Bei-Tseng Chu. 2012. Evaluating Interactive Support for Secure Programming. In *Conference on Human Factors in Computing Systems (CHI)*. 2707–2716. DOI:http://dx.doi.org/10.1145/2207676.2208665