Just-in-Time Static Analysis

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
‡Microsoft Research, USA, livshits@microsoft.com
§Paderborn University and Fraunhofer IEM, Germany, eric.bodden@upb.de
¶North Carolina State University, USA, jssmit11@ncsu.edu and 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 static 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. This paper outlines general guidelines for designing JIT analyses. We also present a general recipe for turning static data-flow analyses into JIT analyses through a concept of layered analysis execution illustrated through Cheetah, a JIT taint analysis for Android applications. Our evaluation of Cheetah on real-world applications and our user study show that JIT analyses are able to present those warnings that are of importance to the code developers just-in-time, allowing them to start fixing problems immediately, without losing their context. Furthermore, study participants consistently reported higher satisfaction levels with Cheetah compared to its traditional counterpart.**

I. Introduction

More companies are integrating static analysis checks in their development process to detect software bugs early enough in the development cycle. However, most static analysis tools, such Microsoft's PREfix/PREfast [1], [2], Facebook's Infer [3]–[5], HP Fortify [6] and Coverity [7], are designed to be used in batch mode. Because analysis runs on real-life projects can easily take hours, companies run static analysis tools at major release points in the product cycle, or as part of nightly builds. In the morning, developers start their day by pouring 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 [8]–[11]. This 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 (or JIT) static analysis concept, in which we advocate the integration of static analysis into the development workflow, allowing code developers to immediately see the impact of their changes in the code, and correct bugs as early as possible. Additionally, by integrating the analysis into the development environment (IDE), more manageable, "digestible" sets of warnings can be reported almost continuously, instead of providing the user with a long list of warnings at the end of the analysis run. We also advocate a different delivery strategy for the results, namely returning simple-to-fix and more certain results first, and using the time the developers take to tackle them to compute more complex results and, perhaps, false-positive-prone results later, while integrating developer feedback.

We propose the idea of a layered analysis, which starts at a program point currently edited by the developer, gradually expanding the analysis scope to encompass methods, classes, files, and modules further away from where the developer is currently focused. Lower analysis layers quickly produce intra-procedural results, expected to yield few false positives. Later layers might find more complex results further out, but also run the risk of higher analysis running times and higher false positive rates.

To concretely illustrate the concepts in this paper, we instantiate the layered JIT analysis with Cheetah, a taint analysis for Android applications. We have conducted an empirical evaluation of Cheetah, along with a user study with 18 participants, in order to evaluate the benefits and shortcomings of our JIT approach, in comparison with a traditional batch-style analysis. To summarize, this paper makes the following contributions:

- It advocates a renewed focus on the interaction between the static analyzer and the user, with the goal of making static analysis more useful to the user.
- It proposes the concept of JIT analysis that interleaves the process of computing analysis warnings with that of the user fixing them.
- It shows how one can convert existing static analyses into 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 evaluates the implementation in detail, focusing on precision, performance, and interactive experiences, the latter evaluated through a user study.

II. Overview

Despite years of work on eliminating false positives, end-user experience, even for the unsound (or optimistic) commercial tools, tends to be overwhelming [12]; this is sometimes called the "wall of bugs" effect. Observing how developers interact with static analysis tools, we highlight

```
1  public class A {
2    void main(B b)
3      String s = getSecret(); // source
4      String t = s;
5      String u = s;
6      sendMessage(s);
7      b.sendMessage(t);
8      leak(u); // sink, leak (A)
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 (B)
18   }
19 }
```

Fig. 1: Running example for a JIT taint analysis.

that: (1) reporting a warning is effectively useless if it is not likely to be examined or result in a bug fix; and (2) even some true warnings are abandoned because they are difficult to deal with [13]. To draw developers' attention to specific high-priority warnings, batch-style tools usually apply post-analysis filtering and ranking of the results [14], [15]. To achieve the same goal, we impose three properties on any sensible JIT analysis: **prioritization**, **responsiveness**, and **monotonicity**. A JIT analysis typically runs in the background of the IDE, and ranks the warnings not only by reporting them *higher* or *lower* in the result list, but also by reporting them *earlier* or *later* in time. This additional dimension helps developers focus only on a subset of warnings, while the JIT analysis is computing further results in the background. This approach of *interleaving* analysis and developer activities reduces the *perceived* analysis latency, which improves the overall usability of a JIT analysis tool. Finally, a JIT analysis does not simply refine an imprecise pre-analysis. The results of a JIT analysis are monotonic in the sense that the later stages of the analysis do not refute earlier warnings, as this would be highly confusing to the developers.

A. Examples of JIT Analyses

Different strategies can be considered to determine what is relevant to the user, i.e. what should be reported first by a JIT analysis. We outline a few concrete examples for expressing different relevance metrics through three different data-flow analyses.

Taint analysis: A taint analysis tracks sensitive data flows from sources to sinks to detect privacy leaks [16], [17]. For the example in Fig. 1, a taint analysis reports two leaks: one from the source on line 3 to the sink on line 8 (labeled with (A)), and another one from line 3 to line 17 (labeled with (B)). The sink call on line 12 is never reached, because line 11 overwrites the tainted variable `x` with non-sensitive information.

When writing code, developer attention is focused on the particular parts of the code that she is editing. Hence,

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

Fig. 2: Example for a JIT API misuse detection.

```
34 void main() {
35    F g = new F(), h = new F(), f = null;
36    g = f;
37    if(...) h = f;
38    x = f.a; (E)
39    y = g.a; (F)
40    z = h.a; (G)
41 }
```

Fig. 3: Example for a JIT nullness analysis.

it is sensible to prioritize warnings by *locality*, i.e., report those warnings that are closest to the user's working set first. If the user is editing the `main` method in Fig. 1, result (A) should be reported first, as it is located in the same method as the edit point. Result (B) can be reported later, since it is located in a different class.

API misuse detection: Analyses that detect misuses of APIs ensure that programs use APIs correctly by verifying that they follow a certain usage protocol [18]. In Fig. 2, the analysis seeks to verify that a cipher is always initialized before a call to `doWork`. Two warnings are reported: (C) and (D) . The former is harder to detect, since the call to `maybeInit` on line 22 may resolve to either of the two implementations of the method. Applying the same ordering by locality as for the taint analysis, a JIT analysis for API misuse detection should find (D) before (C), since finding (C) requires to compute information over three different classes instead of two.

Another strategy based on *confidence* prioritizes (D), as resolving polymorphic calls is more likely to yield false positives compared to monomorphic calls. Similarly, following a strategy based on *computational resources*, CHEETAH decides at runtime to delay the computation of warnings related to 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 can also be computed precisely and quickly.

Nullness analysis: A nullness analysis searches for null dereferences to avoid runtime errors. In Fig. 3, a nullness analysis reports three warnings: (E) on line 38 because `f` points to `null`, (F) on line 39 because `f` and `g` must-alias after the assignment statement on line 36, and (G) on line 40 due to the may-alias on line 37.

While (E) takes minimal computation to find, (F) and (G) require must and may-alias information, respectively. 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, (E) can be returned directly, while alias information is computed to find (F) and then (G).

An advantage of correcting the first warnings early is that the analysis can update the results accordingly. For example, if the developer fixes (E), the other two warnings ((F) and (G)) are naturally fixed as well. This reduces the total number of warnings the user has to consider.

III. JIT Analysis Through Layering

We next discuss how one can turn an existing data-flow analysis into a JIT analysis, fulfilling the properties described in Section II. Our particular approach reorganizes the analysis into multiple *layers*. The goal is to immediately report the most relevant results to the user. Early analysis layers are run first, yielding the first results within a matter of seconds. The following layers enrich the analysis by computing increasingly complex results.

A. A Possible Choice of Layers for Android Applications

In code development, user focus is concentrated around the current edit point. We propose a layered analysis that computes warnings by increasing the analysis scope, i.e., by taking more and more code into consideration, starting with the code the developer recently modified or viewed. We define a set of layers for this strategy, as shown in Fig. 4. *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 needs to be loaded to be able to compute results up to **L3** . *Monotonicity* is to be assured by the internal implementation of each layer: if a given layer cannot confirm a warning within its own scope then it must leave the final decision to later layers.

B. Layered Analysis Examples

Fig. 5 shows the warnings resulting from our example JIT analyses using the layering system above. In the case of a JIT taint analysis, the warning (A) is found at **L1**, as it is a direct leak, while (B) is found after the resolution of the call on line 7. Assuming that classes `A` and `B` are in the same file, (B) is reported at **L4**. In the case of a JIT API misuse detection, results (C) and (D) are returned after the two calls to `maybeInit` on lines 22 and 25, respectively. Assuming that the calls are not in the same package as the `encrypt` method, (C) is returned at **L7** and (D) at **L6**. Since the chosen layer system does not include alias-specific information, the three null dereferences (E), (F) and (G) are reported at **L1**.

L1	Method	In this layer, the analysis performs data-flow propagation in the same method as the current edit point. The analysis pauses at each method call, and the information propagated to those calls is kept in memory to be resolved at later layers.
L2	Class	From L1, the analysis only resolves those method calls whose callee resides in the same class as the current edit point. Data-flows are propagated accordingly.
L3	Class lifecycle	For event-based frameworks like Android, special components such as activities, services, content providers, and broadcast receivers have their own lifecycle. In this layer, data-flows are propagated through a component's lifecycle methods, if the current edit point is in the same class as the method.
L4	File	This layer propagates along calls to callees that are in the same file as the current edit point.
L5	Package	This layer propagates along calls to callees that are in the same package as the current edit point.
L6	Project monomorphic	This layer propagates only along monomorphic calls within the same project.
L7	Project polymorphic	This layer includes further all polymorphic calls within the same project.
L8	Android lifecycle	In the case of frameworks (we consider Android), several components can interact and data can be propagated implicitly through the framework. Such flows are handled at this layer.

Fig. 4: Analysis layering in this paper for analyzing Android apps.

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

Fig. 5: Results reported by each location-based layer for three JIT analyses, when analyzing the Java examples in Figures 1–3., for the following respective starting points (i.e., the currently edited methods): `main`, `encrypt`, and `main`.

C. Layering an Existing Analysis

We next explain how to turn into a layered analysis any existing data-flow analysis that is distributive, i.e., whose flow functions distribute over the merge operator [19]. This constraint eases exposition. In general, one can also apply layering to non-distributive frameworks.

Definitions: We define a *trigger* as a program statement at which the analysis needs to pause the propagation of certain data-flow facts to prioritize others. In the example from Fig. 1, the triggers are the two calls to `sendMessage` on lines 6 and 7. At those triggers, the JIT analysis should delay the propagation of `s` and `t` and propagate `u` first to report (A) in priority. The choice of whether to propagate `s` or `t` next depends on the *priority layers*. The data-flow facts created by a particular layer at a trigger create a *task*. In the same example, the JIT analysis pauses at the triggers, and two tasks are created: one with the initial set `{s}` with priority **L2**, and one with the set `{t}` with

Algorithm 1 Formalization of a JIT analysis

```
1: procedure MAIN
2:     computedTasks = ∅
3:     while PQ ≠ ∅ do
4:         pop task t off priority queue PQ
5:         if t ∉ computedTasks then
6:             ANALYZE(t)
7:             computedTasks ∪= {t}
8: procedure ANALYZE(⟨l, s_t, in⟩)
9:     wl := {s_t} //init worklist
10:    IN[s_t] = in
11:    while wl ≠ ∅ do
12:        pop s off wl
13:        if s is a trigger and s_t ≠ s then
14:            for l' ∈ {1..|layers|} do
15:                in' := {i ∈ IN[s] | layer(s,i,l) = l'}
16:                add new task ⟨l', s, in'⟩ to PQ
17:        else
18:            OLD := OUT[s]
19:            IN[s] := ⊔{OUT[p] | p ∈ preds(s)}
20:            OUT[s] := f_s(IN[s])
21:            if OLD ≠ OUT[s] then
22:                wl ∪= succ(s)
```

PQ returns tasks with the lowest priority layers first.

priority **L4**. The analysis executes the first task - since its priority layer is higher, propagating `s` until the end of the program (or until it reaches the next trigger), and then executes to the second task to report (B).

General algorithm: To illustrate how one can adapt a general, distributive data-flow analysis into a JIT analysis, we present the general JIT-analysis Algorithm 1. The algorithm is designed to require only a minimum number of changes to the analysis solver. The definition of the data-flow analysis problem remains entirely unmodified.

A traditional data-flow analysis would consist of the procedure `analyze`, without lines 13-16. This is a standard fixed-point analysis iterating over the different statements of a program, calling the flow function f_s on those statements (line 20) until the resulting OUT-sets remain unchanged.

The conversion into a JIT analysis divides this large fixed-point iteration into smaller tasks. When reaching trigger-statements within a task, the analysis forces an intermediate fixed-point to be reached by not modifying the OUT-set (line 13). This forces the current analysis task to stop prematurely. All other statements which do not correspond to triggers are handled in the same fashion that the base analysis would normally do (lines 17-22).

To allow the analysis to *eventually* compute the same results as the base analysis, the JIT analysis creates new tasks at trigger statements, and adds them to the priority queue PQ to be executed later (lines 14-16). When a task finishes, the analysis pops the next highest-priority task from PQ. It then creates a new instance of the base analysis to continue propagating where the previous task stopped. The new analysis is initialized at the trigger point where its predecessor stopped, with the appropriate IN-set. The role of the priority queue is to delay or prioritize certain propagation tasks, in order to discover certain warnings first.

When creating new tasks, the analysis uses the `layer(s,i,l)` method to determine the priority layer `l'` to assign to the new task, which will continue propagating the face `i` at statement `s`, knowing that it was paused at layer `l`.

Termination: The JIT algorithm expands on an existing base analysis. Assuming that the base analysis terminates, the inner loop (line 11) is guaranteed to terminate for all analysis instances, as the algorithm does not add to or modify the IN and OUT sets. The outer loop (line 3) also terminates, as the number of potentially created tasks is bounded. Tasks depend on their associated set of facts, and as long as the base analysis' data-flow lattice is bounded, the number of facts — and, therefore, tasks — is also bounded. The check at line 5 ensures that no task is computed twice, providing termination (and improving efficiency).

Soundness: To keep the same level of soundness as the original analysis, the layers of the JIT analysis have to ensure that at each statement, every data-flow fact created by the base analysis' flow function should be assigned to at least one layer. Additionally, on line 15, the algorithm partitions the statement's IN-set into smaller sets. For this operation to be safe, the data-flow facts should be separable, i.e., the analysis problem should be distributive. In this case, the data-flow facts can be independently distributed between the layers and we can improve efficiency by requiring that each data-flow fact of an IN-set should be assigned to one layer, and one layer only.

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 should be carefully chosen to always provide a complete and disjoint partitioning of the IN-set between the different layers.

The layering described in Section III-A fulfills the above criteria rather trivially by using only method calls as triggers and by partitioning IN-sets according to the callees to which they are passed. Other layerings are possible, however. For instance, one might define layers distinguishing data flows with or without aliasing.

IV. CHEETAH: A JIT Taint Analysis

Following the layered approach from Section III, we have implemented Cheetah, a JIT taint analysis that detects data leaks in Android applications. We have also integrated Cheetah into the Eclipse Integrated Development Environment (IDE) to be able to evaluate its usability with a user study. Cheetah is built on top of the Soot analysis framework [20] and the Heros IFDS solver [21]. Cheetah is based on a simple taint analysis that tracks explicit dataflow. We use IFDS [19] as a succinct way to

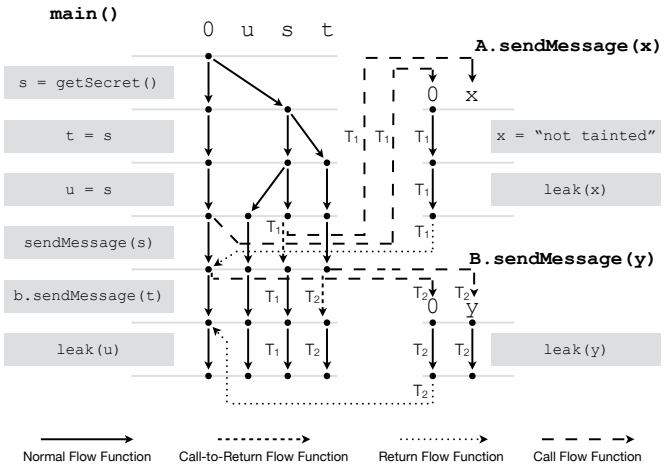

Fig. 6: IFDS taint analysis for the example in Fig. 1. T_1 and T2 mark the edges created by CHEETAH for tasks T_1 and T_2, respectively. The unmarked edges are created for task T_0.

capture out taint analysis. IFDS is designed to solve interprocedural 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.

Data-flow domain: An IFDS analysis' data-flow domain is composed of data-flow *facts* (or abstractions) that might hold at each statement of the analyzed program. A tautological fact, `0`, holds at each statement of the program and is the root of the exploded super-graph. For a taint analysis, the data-flow domain D is composed of all references present in the program. We model them as *access paths*, i.e., a combination of a local variable and successive field accesses. For example, the access path $x.y.z$ denotes the field $\lambda = y.z$ of the base object x. A fact holds if it is reachable from the root of the exploded super-graph: the original `0`. In our taint analysis, a fact holds if the corresponding access path is tainted.

Flow functions: In the IFDS framework, the flow functions map each existing data-flow fact to its successors (D to 2^D). For a taint analysis, a typical flow function would (1) generate new taint flows if it encounters source methods; (2) kill taints if the tainted variable is overwritten by non-tainted data or (3) transfer taint if tainted references are assigned to other references. Generally, IFDS uses four different kinds of flow functions, which we illustrate for the running example from Fig. 1 in Fig. 6.

- **Normal flow functions**: are applied at each statement that is not a call. For example, transferring the taint from s to t on line 4 is handled by a normal flow function. Note that s is also transferred to s since after the statement s is still tainted.
- **Call flow functions**: are applied at each call statement. They transfer the data-flow facts from the caller scope to the scope of the callee. For example, on line 7, t is mapped to y.
- **Return flow functions**: are also applied at each call statement and behave inversely to the call flow functions, mapping the facts back from the callee to the caller.
- **Call-to-return flow functions**: are also applied at each call statement. They transfer the facts that are not affected by the call. For example, on line 7, u is mapped to u. Sinks are also handled in the call-to-return flow function. If the method m is a sink, and if it leaks the tainted variable α, it is reported. For our implementation, we use the sources and sinks described in Rasthofer et al. [22].

A. Base Taint Analysis

We next present the flow functions of the base IFDS analysis. We denote by $\langle stmt \rangle(\alpha)$ the flow function of $stmt$ applied to an access path α. The taint analysis is based on the Jimple intermediate representation [23], which eases the analysis, as it is a three-address code representation. We simplify our presentation by assuming only single call parameters and by only considering assignment and call statements, as the other statements are irrelevant to taint analysis.

Normal-flow function:

$$\begin{aligned} \langle x \leftarrow y \rangle(\alpha) &= \{\alpha\} \setminus \{x.*\} \cup \{x.\lambda \mid \alpha = y.\lambda\} \\ \langle x \leftarrow y \otimes z \rangle(\alpha) &= \{\alpha\} \setminus \{x.*\} \cup \{x \mid \alpha = y \vee \alpha = z\} \\ \langle other \rangle(\alpha) &= \{\alpha\} \end{aligned}$$

The normal-flow function is the identity function except for assignment statements where one of the right-hand side operands is tainted. For a direct assignment, we remove all taints for the left operand (x and all of its fields), and apply to its base variable the taints that exist for the right operands and its fields ($x.\lambda$). In the case of a binary operator $\otimes$ Jimple requires all operands to be local varables and thus we simply taint the left operand if any of the right operands are tainted.

Call-flow function:

$$\langle x \leftarrow a.m(p) \rangle(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \text{ is static} \vee \alpha = 0 \\ \{this.\lambda\} & \text{if } \alpha = a.\lambda \\ \{arg.\lambda\} & \text{if } \alpha = p.\lambda \\ \emptyset & \text{otherwise} \end{cases}$$

For the call-flow function, taints for the base variable and the parameters are propagated into the callee. They are mapped to the *this* variable and formal-argument name *arg* in the scope of the callee. Static variables are propagated as well.

Return-flow function:

$$\langle x \leftarrow a.m(p)\rangle(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \text{ is static } \vee \alpha = 0 \\ \{a.\lambda\} & \text{if } \alpha.\lambda = this.\lambda \\ \{p.\lambda\} & \text{if } \alpha.\lambda = arg.\lambda \wedge |\lambda| > 0 \\ \{x.\lambda\} & \text{if } \alpha.\lambda = retVal.\lambda \\ \emptyset & \text{otherwise} \end{cases}$$

The return-flow function maps the *this* variable and arguments back to the base variable and parameters of the caller scope. Static variables are propagated back into the caller.

Call-to-return-flow function:

$$\langle x \leftarrow a.m(p)\rangle(\alpha) = \begin{cases} \emptyset \text{ if } \alpha \text{ is static} \vee \alpha.\lambda = p.\lambda \vee \\ \qquad \alpha.\lambda = a.\lambda \\ \{\alpha\} \setminus \{x.*\} \cup \{newTaints(m) \mid \\ \qquad \alpha = 0 \wedge isSource(m)\} \\ \qquad \text{otherwise} \end{cases}$$

The call-to-return-flow function ensures that the variables that are affected by the call are not just propagated across the call on the side of the caller. Static variables, parameters, the base variable, are killed by returning $\emptyset$. Their taint, as well as that of the overwritten variable x, may be carried over by the corresponding return flow function, depending on the implementation of the callee(s). The function also propagates further such access paths not referenced by the call. If the method m is a source method, the newly tainted variables are added to the tainted set.

Sinks are also handled in the call-to-return flow function. If the method m is a sink, and if it leaks the tainted variable α, it is reported. For our implementation, we use the sources and sinks described in Rasthofer et al. [22].

B. Layering the Taint Analysis

This paragraph details how we modify the base taint analysis into a JIT analysis using the layers defined in Section III-A. In order to use the general algorithm presented in Section III-C, we define the following: $isTrigger(s) = s.containsMethodCall()$ and $layer(s, i, l) = distance(s.callee, startPoint)$. The JIT taint analysis should be paused at every call site, so we mark method call sites as triggers. The propagation at those call sites is continued in a subsequent task. The layer that is assigned to a fact at a call site is determined by the distance between the callee and the start point of the analysis, which is the currently edited method. The distance is defined in terms of the chosen layers (see Section III-A). For example, if the analysis encounters a call to a method that is in the same file but not the same class as the starting point, the new task would be assigned **L4**. As a result, one task creates as many tasks as the number of call sites it contains. All of those new tasks are added to the priority queue and executed in order of distance to the starting point. To adapt the algorithm to the IFDS framework, we apply the following changes:

- Every time a task is executed (line 6 in Algorithm 1), the analysis creates a new IFDS instance starting at the task's start statement, and initializes it with the facts contained in the task's `inSet`. To reuse the previously computed results, the state of the IFDS solver is carried over from one IFDS instance to the next.
- The priority queue is initialized with the task `{L1, stmt, 0}`, where `stmt` is the first statement of the currently edited method.
- To pause the analysis at call sites and create a new task, we overwrite the base analysis' call-flow function:

$$\langle stmt\rangle'(\alpha) = \begin{cases} \langle stmt\rangle(\alpha) & \text{if } stmt = task.startStmt \\ \emptyset & \text{otherwise} \end{cases}$$

  Returning $\emptyset$ ensures that the propagation of the data-flow facts is stopped at all call sites, except when the call is the start statement of the current task. This corresponds to lines 13–16 in Algorithm 1. In addition to modifying the out-set, the call-flow function also creates new tasks. When the analysis is paused at a call site, it collects the variables that need to be propagated further (i.e., the parameters of the call, the static variables, and the base variable of the call) into an `inSet`. The analysis then creates a new task `{layer(stmt), stmt, inSet}`, and adds it to the priority queue to be executed later. Those taints will be resolved when the analysis processes the call of the newly created task.
- The normal, return, and call-to-return flow functions remain the same as the base analysis.

Example: Applying CHEETAH to the example in Fig. 1 results in the following steps (also shown in Fig. 6):

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) Result (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 with T_1.
4) T_2 is executed, resulting in the edges labeled with T_2, and (B) is reported.

V. IMPLEMENTATION DETAILS

A. Analyzing Incomplete Code

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 following the call graph from edit points. Each task instance implements the methods `requiredTasks` and `nextTask`. The method `requiredTasks` returns a list of all tasks with the same scope (in terms of

layers) as the current task, but with lower layer values. For example, the required tasks of a class-specific task in **L2** would induce the creation of tasks in **L1** for each method of that class. Before executing a task CHEETAH then checks that all of its required tasks have already been executed. The method `nextTask` returns the task that should be executed next, which is a duplicate of the current task, with a higher layer value. For example, a task in **L1** is followed up by a task in **L2** starting at the same method. This is required if the current task does not naturally create any further tasks. This approach enables CHEETAH to help software developers reason about unreachable code, a property that a traditional IFDS-based taint analysis does not provide.

B. Class Loading and Call Graph Construction

Traditionally, an IFDS-based analysis requires access to all classes in a given program. It also requires a call graph as input, which is typically constructed before the analysis starts. Implementing CHEETAH using this traditional approach would result in an unnecessary initial cost for class loading and call-graph construction, which affects how responsiveness CHEETAH is to user interactions. We follow a more suitable approach for CHEETAH by only loading the classes that are necessary to execute the current task by calling the Soot method `Scene.v().forceResolve()` on each of those classes. To construct the call graph, CHEETAH uses Soot's `OnTheFlyJimpleBasedICFG`, which is an on-demand algorithm that uses the class hierarchy to resolve calls, similar to Class Hierarchy Analysis (CHA). Our approach for class loading and call-graph construction enables CHEETAH to quickly deliver the results that are computed in the early layers.

C. Android Lifecycle

Unlike Java programs, Android applications do not contain a main method to start the analysis from. Various callbacks create implicit data flows that should be modelled by a sound analysis. Special Android components such as activities, services, content providers, and broadcast receivers have their own lifecycle, which is expressed by overwriting lifecycle methods. The Android framework calls those methods to start, stop, pause, resume, and destroy the component.

To emulate the Android lifecycle and callbacks, Arzt et al. [24] introduce a dummy main method that explicitly calls all registered callbacks in all possible orders. However, this approach is not useful in the context of a JIT analysis such as CHEETAH, because resolving callbacks in a class requires loading it. Therefore, creating the dummy main method for the whole application means loading all classes when CHEETAH is launched. This contradicts the class-loading and call-graph construction strategy that CHEETAH uses.

In CHEETAH, we thus defer the creation of the dummy main method to the last layer **L8**, when all the other tasks have been executed. However, a component's lifecycle methods are typically declared in the same class, and resolving such flows at the last layer is counter-intuitive, especially when the component's class is loaded as early as layer **L2**. As a result, in layer **L3**, CHEETAH creates a component-specific, local dummy main method. Leaks at this layer are resolved early, and only inter-component flows are delayed to layer **L8**. The global dummy main created at that last layer then calls the previously-created local dummy main methods, instead of directly referring to the callbacks.

D. Reporting Results

To report results useful to software developers, CHEETAH provides a witness for each reported warning. To track tainted paths, we augment our access paths with more information, similar to Lerch et al. [25]. Each data-flow fact holds its predecessor, its source statement, and a list of neighbours. A loop-aware depth-first search then provides one or more witnesses that show the path causing the warning reported by CHEETAH. We also use those witnesses to compute locality metrics for our empirical evaluation.

VI. EMPIRICAL EVALUATION

We empirically evaluate CHEETAH by comparing it to BASE, a traditional IFDS-based taint analysis through answering 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?
RQ4: What is the *precision* and *recall* of CHEETAH and BASE compared to FLOWDROID?

A. Experimental Setup

Both CHEETAH and BASE are implemented on top of the Heros IFDS solver [21], based on Soot [26], and share the same flow functions. Our benchmark consists of 14 Android applications selected from the most recent 100 applications in the F-Droid repository [27], such that each application has a GitHub repository with more than one commit. We then used Boa [28] to mine those repositories and determine the methods that have been modified in each commit. We then ran two experiments per application. In the first experiment, we used the modified methods in GitHub commits as the starting points for CHEETAH (hereafter, referred to as SPB).

Each application has at least 26 unique SPB (min: 26, max: 316, median: 127). We then ran CHEETAH once for each application, using 20 randomly selected SPB, to obtain the actual sources and sinks. In the second experiment, we ran CHEETAH using those actual/known sources as starting points for the analysis (hereafter, referred to as SPS). The starting point of BASE is always the dummy main method that it creates for the Android

application under analysis. 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 1,024 MB.

B. 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 compared them to those when it starts at SPS. We have also compared these times to the time it takes BASE to report its final results. Fig. 7 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" [29]. CHEETAH reports its last result in a median time of 9.16s and 7.78s when it starts at SPB and SPS, respectively. Both medians are larger than the median time of 2.13s that BASE takes to report its final results. This is attributed to the fact that CHEETAH is capable of analyzing parts of the program that are not reachable from its main entry points, a feature that traditional analyses such as BASE do not offer. We discuss in **RQ4** why this feature is desirable for real-life scenarios. Furthermore, using CHEETAH, enables the user to either continue her development tasks, or addressing the first warnings that she gets.

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 help software developers detect bugs located around their working set faster compared to using traditional analyses. This means that CHEETAH should ideally report most of its warnings in earlier layers. Fig. 8 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**. In such a case, CHEETAH reports more warnings in earlier layers: a median of 4.58% in **L1** and a median of 5.13% in **L3**. This shows that if CHEETAH is guided towards the points of interest in a program, more warnings are reported earlier. Therefore, starting at SPS is more optimal when a user requires analysis updates, while fixing a particular warning.

The process of computing the witness that accompanies the reported warning 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 while computing the warnings. 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 does not currently support inter-component flows.

Cheetah reports most of the warnings in **L5** and **L6**. If directed to known sources of bugs in a program, Cheetah 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 evaluate 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. Fig. 9 shows the trace lengths for the warnings that appear in each layer of CHEETAH. When CHEETAH starts at SPB, the length of the traces for the initial layers **L1**-**L4** is a median of 0, 1, 3, and 4 statements, respectively. For later layers, more complex warnings are reported. Therefore, the median length of the traces for layers **L5** and **L6** is 26 and 22 statements, respectively. Starting at SPS enables CHEETAH to report warnings at 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, 1, 11, and 1 statements, respectively.

The initial leaks reported by Cheetah have shorter traces, making them easier to interpret.

RQ4: What is the *precision* and *recall* of Cheetah and Base compared to FlowDroid? We compare the precision and recall of CHEETAH and BASE to FLOWDROID, a highly precise taint analysis for Android applications [24]. We use FLOWDROID as our ground truth to verify that both CHEETAH and BASE report correct results. The Venn diagram in Fig. 10 depicts the results found by each analysis for our benchmark applications. Each point in the Venn diagram, indicated by both a color and a number, represents a specific result that is found in one application. Both BASE and CHEETAH find all the warnings that are reported by FLOWDROID, except for four warnings that require handling threads and modeling application layout, which are both currently not supported in either tools. Across our benchmark, BASE is less precise than FLOWDROID as it reports $2.5x$ more warnings (min: $1.5x$, max: $8x$, geometric mean: $2.54x$). Since BASE and CHEETAH have the same IFDS flow functions, they both should have the same precision and recall when compared to FLOWDROID. However, CHEETAH reports $3.6x$ more warnings compared to FLOWDROID (min: $1.5x$, max: $28x$, geometric mean: $3.58x$). We have further inves-

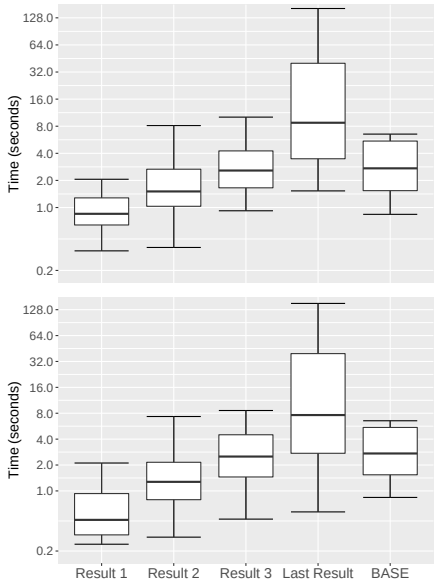

Fig. 7: Time to report results (in log scale) for CHEETAH compared to BASE when starting at SPB (top) and SPS (bottom).

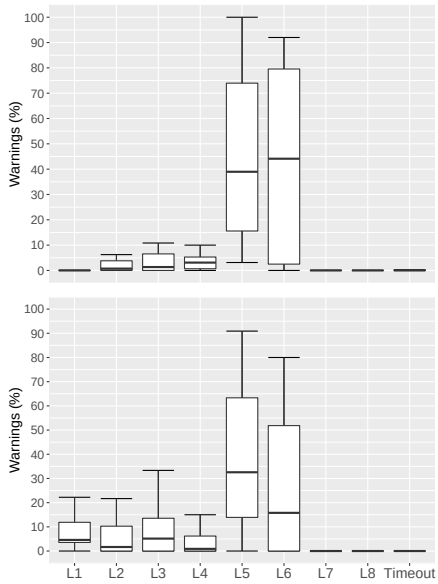

Fig. 8: Percentage of warnings that appear in each layer when CHEETAH starts at SPB (top) and SPS (bottom).

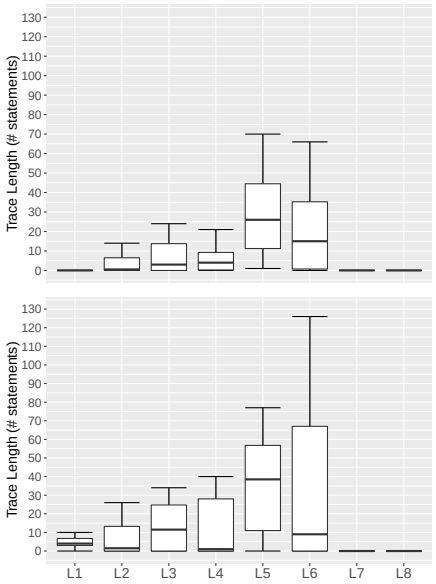

Fig. 9: The length of the traces for the warnings that appear in each layer when CHEETAH starts at SPB (top) and SPS (bottom).

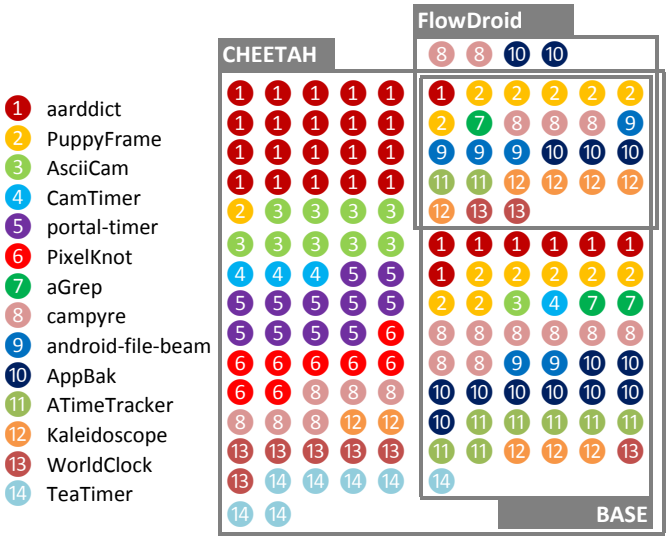

Fig. 10: Analysis results and their intersections as a Venn diagram.

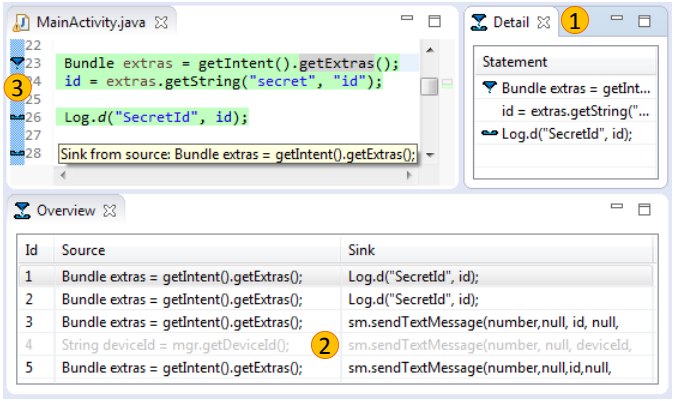

Fig. 11: GUI elements of CHEETAH.

Base and Cheetah have the same precision and recall compared to FlowDroid. However, Cheetah can analyze parts of the program that are more relevant to the user.

tigated the issue, and confirmed that all of these warnings stem from the fact that CHEETAH is capable of analyzing parts of the application that are unreachable from the program entry points. In real life, developers may be working on some code that is still unreachable from the program entry points. CHEETAH analyzes those parts of the codebase to provide the user with a more relevant result set than traditional analyses such as BASE. To achieve that, we create artificial tasks that guide CHEETAH to analyze those parts of the program, and add them to the priority queue that CHEETAH processes.

VII. GRAPHICAL USER-INTERFACE

Reporting results while the code is modified and the analysis still runs in the background requires a careful handling of warnings. If no care is taken, warnings literally become moving targets, shifting around in result lists as new warnings are found and others are fixed, becoming confusing for the developer. Fig. 11 illustrates the GUI features of CHEETAH, which have been introduced to support reporting warnings over time. These features, described below and highlighted with numbers in the figure, were inspired by the observations made during our pilot study.

1. **Views:** To avoid overwhelming the users by showing them all warnings and their details information in one place, we separate the warning information into two

views: *overview* and *detail*. The first view provides a list of all reported data leaks, while second view presents the path details of the warning selected in the overview view. The detail view provides a compact summary of the leak, offloading the amount of information contained in the overview view.

2. **Color-coded warnings:** Warnings in CHEETAH can have three states: *active* (confirmed by the latest analysis run), *computing* (found by the previous run of the analysis, and not yet confirmed by the current run), and *fixed*. We display active warnings in black and computing warnings in gray. Fixed warnings are kept grayed out for one run of the analysis, and removed from the view at the next run, thus providing a light history of fixed leaks. This allows 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 in the overview view is grayed out, the corresponding icons are also grayed out. Tooltips provide additional information on the particular leaks the icons refer to.
4. **Other features:** A few features that are not JIT-specific have been provided for the sake of clarity, such as highlighting the path of the leak the user is currently examining, and introducing unique identifiers to help users identify the leaks they are interested in.
5. **Seamless run:** To integrate the tool into the development environment, we chose to trigger the tool every time the project is built. CHEETAH hooks into Eclipse's incremental builder, and re-runs the analysis starting from the method that currently has the focus.

VIII. USER STUDIES

In order to evaluate how a JIT analysis integrates into the development workflow compared to a traditional batch-style analysis, we conducted a user study using CHEETAH and BASE.

We first conducted a pilot study with 11 participants of mixed skill and experience levels, in both Android development and static analysis tools (including professional developers and Android security analysts). The outcomes of this study were mixed, as most participants were somewhat distracted by our initial GUI prototype for the JIT analysis. One specific issue was the overview view that kept changing as new results were computed, making it impossible to track a particular warning, and for which we introduced the graying system presented in Section VII. We fixed the issues reported by the initial batch of participants and repeated the study, with better results.

Participants: The actual study included 18 participants of varying backgrounds (9 academics and 9 professional code developers), and of 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].

Task: To make the conditions of the study as close to day-to-day development activities as possible, the participants were asked to perform a development task: removing code duplicates in an Android applications. Their secondary goal was to keep the number of *data leaks* to a minimum, while they were coding. CHEETAH or BASE were provided as Eclipse plugins to help them with detecting potential data leaks. Data cleansing/sanitization APIs were provided to prevent potential leaks.

Protocol: The participants performed the task a total of *four* times. They switch the Eclipse plugin in the middle, after the first two times. We randomized the use of plugins, so that half of the participants started with CHEETAH and the other half, with BASE. For each tool, the participants first worked with a small Android application to get used to the IDE and the plugin. They then proceeded on to a bigger real-life application. Afterward, we asked the participants to fill a comparative survey, and interviewed them in person. The full list of survey questions, participants' responses and the interview protocol are provided online[2].

Test applications: The same two applications (small and big) were used for both tools across all participants. The small one is an artificial Android app designed to showcase the tool's features. The real-life application is Bites, a basic cookbook app, taken from F-Droid[3]. Because each task had to be completed in a limited amount of time (10 minutes), we modified the application to arrange for some code duplications around *existing* data leaks. This effectively forced the users to interact with the analysis tools more often, which is what lacked in the pilot study. Unlike the pilot study, we also chose to keep the same two applications for both tools, so that the users would not need to relearn the code itself, spending more time using the plugins.

GUI: To avoid a GUI-induced bias, the GUI used for BASE is almost identical to CHEETAH's. For this user study, BASE analysis emulates a batch-style tool, similar to HP Fortify's Eclipse plugin [6]. The analysis is not triggered on code build, but instead by pressing a button. A popup then blocks the GUI, to prevent the user from modifying the code, while the analysis is executing. All results are shown at the same time, after the analysis finishes. The warnings are displayed in the order in which they are found. For both tools, we disallowed participants to reorder the warnings presented in the warning view, to evaluate the efficiency of CHEETAH's built-in ordering strategy.

[1]We ignore **P17**, as (s)he used the analyses in a manner that triggered a UI bug, and was unable to properly perform the tasks.

[2]`https://blogs.uni-paderborn.de/sse/files/2016/08/JITA_UserStudy.pdf`

[3]`https://f-droid.org/wiki/index.php?title=caldwell.ben.bites`

A. Survey Results

Format: After the development tasks, the participants filled a survey comprised of 29 questions designed to assess the merits of the two approaches. In the survey, participants provided some open-ended comments and compared CHEETAH to BASE. Participants also answered several 5-point Likert-type questions from the System Usability Scale (SUS) [30], a questionnaire designed to measure the effectiveness and efficiency of a system. Finally, participants rated both tools using a Net Promoter Score (NPS) [31], a 11-point Likert scale measuring their likelihood of recommending the tool to a friend.

Results: Overall, participants responded positively to CHEETAH. Among participants who rated both CHEETAH and BASE on the NPS question ($n = 16$), CHEETAH's mean score was 7.4 (out of 10) as compared with 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 [32], we observed significant ($p < .05$) differences between these aggregated scores and participant's responses on 4 of the individual SUS questions. Compared with BASE, participants were less likely to find CHEETAH unnecessarily complex (-.6 mean response), or cumbersome (-.6 mean response). Further, participants responded that they were more likely to use CHEETAH frequently (+.7 mean response) and were more likely to find its functions well-integrated (+.5 mean response).

B. Interview Results

Format: After filling the survey, participants were interviewed individually. We asked them to detail their experience of the tools, focusing on the perceived differences, in particular concerning waiting times, integration of the tools in the IDE, and warning ordering. The interviews lasted 14 minutes in average, ranging from 10 minutes to 23 minutes. We present below the notable comments and behavior observed during the overall study, organized by user interface features.

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 was "much more comfortable, and what I would expect in the Eclipse environment" (**P7**). As a result, after starting the analysis, those 12 participants resumed their tasks as soon as the update they expected was confirmed. The other 5 participants did not notice or act upon seeing the updated warnings, which we detail in a later paragraph.

Integration in the development workflow: Participants noted that CHEETAH's seamless run feature made it better integrated into Eclipse: "it integrates well into the Eclipse build-on-save paradigm" (**P2**). This resulted 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: Seven participants complained about the random ordering offered by BASE, saying that they would like to be able to "sort the leaks by source, sink, line number" (**P6**). No such complaints were made for CHEETAH, regardless of the order in which the participants used the tools, which indicates that the ordering provided by the chosen layers was not harmful to the participants' performance. **P8**, a professional Android developer, "felt like there was an ordering by class for [Cheetah]", and commented: "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". **P18** in particular, dealt with the leaks in the proposed order. **P18** fixed all 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 our priority layers positively affects participant performance and integrates more discretely in their development 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**). Also, 5 participants felt "the analysis was slowing the IDE down" (**P13**), as it took more time to compute all results than BASE. We note that CHEETAH is fully computed in the background, and was given enough memory to not interfere with the UI thread.

Waiting times: During the study, five participants waited until CHEETAH terminated before starting to look at the results. One participant wanted to have a full overview of all warnings at each code change. Another one started fixing leaks starting from the bottom of the list; those were always reported at the latest layers. The other three participants were frequent users of blocking analysis tools. They automatically chose to not touch the code while CHEETAH was computing, wanting to "avoid errors when intermediate results are computed [...] while the analysis is in an unclean state" (**P15**).

Graying system: The graying system was deemed useful by some participants: "I had the impression I did something" (**P16**), "History is useful" (**P9**), but it proved confusing to others, as they still had trouble finding out if a fix was successful. Some noted that the "color of the fixed issue should be different [than a computing issue's]" (**P4**), others, that it was "disturbing because results changed" statuses during the analysis (**P6**). This prompted them to wait until the overview view (or the part of the view where their warning was located) stabilized. **P1** noted that the feedback given

by the tool on a change "should be more prominent". To see if a leak was fixed, some participants relied on the gutter icons instead: "The sink [icon] has now disappeared, that's a good thing" (**P5**).

Comparison: During the interviews, we asked the participants in which cases CHEETAH would be more useful than BASE and vice-versa.

- 12 participants reported CHEETAH would be 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". 2 participants expressed concerns 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**).
- 11 participants noted that BASE should be used infrequently, for example, "after a milestone" (**P9**), or in situations where debug and coding are separated, such as "creating reports for software" (**P7**). No participants reported they would use BASE for code development.

12 out of 17 participants preferred Cheetah for code development due to its quick updates, although concerns were expressed about CPU overhead and UI details. Cheetah's inherent ordering also helps code developers for bug fixing tasks.

IX. LIMITATIONS AND FUTURE WORK

Incremental Analysis: Previous approaches at improving the responsiveness of an analysis include incremental analyses, which only compute analysis changesets at each code modification introduced by the user. A full analysis is required from time to time, especially when the user makes impactful changes to the code. In those cases, the analysis remains as slow as a standard analysis, while the performance of a JIT analysis remains consistent over all runs. The main drawback of CHEETAH is that it fully recomputes the whole analysis every time it is rerun. Incrementalizing CHEETAH would effectively address this drawback, and result in an analysis that is fully responsive, while still providing the code developer with useful results first.

Layering System: As mentioned in Section III-C, priority layers should be chosen carefully to provide a complete and disjoint partitioning of the base analysis' in-set at any statement. Defining a correct and useful set of layers is not an easy task, and we plan to improve on how to do such a thing.

Non-distributive problems: Section III-C shows how to lift a base analysis into a JIT analysis. Because facts are separated at line 15, the analysis problem must be distributive in order to use the proposed algorithm. Changes could be added to the algorithm to support non-distributive problems, which we plan to include in future work.

X. 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.

A. Human Aspects of Static Analysis Tools

Several researchers have conducted studies of developers' usage of static analysis tools. Sadowski et al. [33] found that most Google developers reported that static analysis warnings were usable. Phang et al. [34] found that a program flow visualization tool helped developers quickly triage warnings. Ayewah and Pugh [35] found that checklists helped developers consistently evaluate warnings. In an experiment, Smith et al. [10] 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. [8] recorded interviewees stressing the importance of integrating static analysis into their workflows. Lewis et al. [11] found that almost all interviewed developers agreed that static analysis should not disrupt their workflow. Through interviews and surveys, Xiao et al. [36] and Witschey et al. [37] found that developers whose security tools help them do their work quickly report being more likely to adopt those tools. Christakis and Bird [38] 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.

B. Warning Prioritization

Researchers have proposed several ways to prioritize which static analysis warnings developers should address first. Industrial tools tend to use heuristics, such as in FindBugs [39], which classifies warnings as low, medium, or high priority. Surveying the research, Muske and Serebrenik [40] organize prioritization approaches into three main categories: statistical, historical, and user-feedback. As an example of a user-feedback based approach, Heckman and Williams [41] use machine learning to prioritize *actionable* warnings over *unactionable* ones. As an example of a history-aware approach, Kim and Ernst [42] use code history to prioritize defects. Other approaches do not easily fit into these categories. For example, Shen et al. [43] deprioritize predicted false positives, then use developer feedback for future prioritization. As another example, Liang et al. [44] 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.

C. Incremental Presentation

Several prior researchers have investigated how to incrementally present analysis results to the user. For example, Parfait [14], [15] 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. On the other hand, 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 workflow.

CHEETAH reports warnings in a similar way to how Eclipse's incremental compiler reports errors to a user while editing source files [45]. This is the same approach used by ASIDE [46], an Eclipse plugin that detects security vulnerabilities in Java programs. ASIDE incrementally reports errors to the user by only analyzing recent code changes. Although CHEETAH is a whole-program analysis, it still incrementally reports warnings to the user by starting at specific points of interest in the program (e.g., a recently modified method). However, we plan to introduce incremental analysis to CHEETAH in the future to improve the responsiveness of its later layers **L5**-**L8**.

XI. CONCLUSION

We have presented the novel concept of JIT analysis, which interleaves the processes of code development, static analysis execution, and bug fixing, by layering the static analysis approach. We have shown how to obtain a JIT analysis by modifying a base distributive data-flow analysis with minimal changes, using a layering system. CHEETAH, an implementation of a JIT taint analysis for finding privacy leaks in Android apps, has been evaluated on real-world Android benchmarks. Our empirical results, survey 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] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30(7):775–802, 2000.

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

[3] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[4] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer, 2011.

[5] Infer. A static analyzer for mobile apps. `http://fbinfer.com`.

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

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

[8] 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 *ICSE*, pages 672–681, 2013.

[9] Nathaniel Ayewah and William Pugh. The Google FindBugs fixit. In *ISSTA*, pages 241–252, 2010.

[10] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[11] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *ICSE*, pages 372–381, 2013.

[12] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[13] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[14] Cristina Cifuentes. Parfait - A scalable bug checker for C code. In *SCAM*, pages 263–264, 2008.

[15] 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.

[16] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, 2010.

[17] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.

[18] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *Onward!*, pages 1–13. ACM, 2015.

[19] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[20] 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 *CC*, pages 18–34, 2000.

[21] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP*, pages 3–8. ACM, 2012.

[22] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[23] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[24] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[25] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 619–629. IEEE Computer Society, 2015.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, 1999.

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

[28] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE Computer Society, 2013.

[29] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[30] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[31] Frederick F Reichheld. The one number you need to grow. *Harvard business review*, 81(12):46–55, 2003.

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

[33] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.

[34] Yit Phang Khoo, Jeffrey S Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 57–63. ACM, 2008.

[35] Nathaniel Ayewah and William Pugh. Using checklists to review static analysis warnings. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 11–15. ACM, 2009.

[36] Shundan Xiao, Jim Witschey, and Emerson R. Murphy-Hill. Social influences on secure development tool adoption: why security tools spread. In *CSCW*, pages 1095–1106, 2014.

[37] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271, New York, NY, USA, 2015. ACM.

[38] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study.

[39] Findbugs. `http://findbugs.sourceforge.net`.

[40] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *SCAM*, 2016.

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

[42] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[43] Haihao Shen, Jianhong Fang, and Jianjun Zhao. Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 299–308. IEEE, 2011.

[44] Guangtai Liang, Qian Wu, Qianxiang Wang, and Hong Mei. An effective defect detection and warning prioritization approach for resource leaks. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 119–128. IEEE, 2012.

[45] Andrew Jensen Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *CHI*, pages 387–396. ACM, 2006.

[46] Jing Xie, Heather Lipford, and Bei-Tseng Chu. Evaluating interactive support for secure programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2707–2716, New York, NY, USA, 2012. ACM.