A Scalable, Flow-and-Context-Sensitive Taint Analysis of Android Applications.

Wontae Choi*, Jayanthkumar Kannan[1],*, Domagoj Babic*

Google Inc. 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA

Abstract

This paper focuses on scalable static analysis techniques for finding information leaks in Android apps. Finding such leaks scalably is challenging because Android apps have on average over 100 invocations of sensitive APIs, yielding a massive multi-source taint analysis problem.

We present the design of *STAR*, a context-sensitive and flow-sensitive multi-source taint analysis aimed at tackling this problem. *STAR* incorporates two main ideas to achieve high performance and scalability. The first is a novel summarization technique we refer to as symbolic summarization, which is crucial for the analysis to scale well with the number of source APIs. The second is a combination of techniques aimed at efficient propagation of abstract states both within and across method boundaries. Our experiments over a dataset composed of $400,000$ apps show that the proposed techniques improve performance over an IFDS-style analysis by a factor of 30 on average, and by up to four orders of magnitude on large apps.

Keywords: Android, taint analysis, summarization, scalability

Declaration of Interest

We confirm that the manuscript has been read and approved by all named authors. The work has been supported by Google Inc. The manuscript has been reviewed and approved via the internal review process of our institution, and there are no impediments to the submission with respect to intellectual property.

*Corresponding authors: Wontae Choi and Domagoj Babic

Email addresses: `wtchoi@google.com` (Wontae Choi), `dbabic@google.com` (Domagoj Babic)

[1]The work has been done while the author was working at Google.

1. Introduction

Smartphones are an important part of modern life, and so is the problem of smartphone security. Users entrust their smartphone apps with personal and private information. Deliberate or mistaken release of that data to untrusted parties without users' consent is highly undesirable and can significantly impact the trust users place in the security of the entire ecosystem. Therefore, app stores have strong incentives to prevent such undesired data leakage.

Finding apps that expose user data is a hard problem. App stores typically employ a variety of techniques such as dynamic analysis, machine learning, and static analysis. This paper focuses on static taint analysis techniques to find such information leaks in the context of the Android ecosystem.

In the Android app store setting, there are three aspects of the information leak problem that make it challenging. First, the number of relevant sensitive APIs in the Android framework is large, resulting in a massive multi-source taint analysis problem. In our evaluation corpus, Android apps had 115 invocations of sensitive APIs on average (the 99th percentile is 1244). Second, an app uploaded by a developer has to be analyzed before it is allowed to be published on the store. To minimize the app publishing delay, the analysis time is severely limited. Third, the scale of the problem is immense. Popular app stores have over a million apps, many of which are updated daily. The size of apps varies from a thousand to a few hundred thousand classes.

While taint analysis has received considerable attention (*e.g.*, [3, 14, 21, 41, 37]), our focus is specifically on the *multi-source* setting. We focus on the design and evaluation of taint analysis algorithms that can handle multiple sources efficiently and scale well to a production system with strict time budgets and resource constraints.

We present *STAR* — **S**calable **T**aint **A**nalyzer for And**R**oid — a context-sensitive and flow-sensitive multi-source taint analysis. *STAR* is an instance of the *interprocedural, finite, distributive, subset framework (IFDS framework)* [34] and we describe its design in terms of abstract semantics. The analyzer is capable of analyzing 77% of $400,000$ Android apps sampled from Google Play Store within a per-app time budget of four hours. More generally, the techniques we propose are applicable to any Java program. We make three novel contributions in the design and evaluation of *STAR*.

The first contribution is a novel summarization technique we refer to as symbolic summarization. This is inspired by the *Reps-Horowitz-Sagiv tabulation algorithm (RHS algorithm)* [34] and polymorphic type-and-effect systems [22]. The technique is based on two key insights: (a) the symbolic method summary is parametric, which allows the analysis to avoid performing redundant computation, and (b) the summary captures global state changes (sensitive data propagated to sinks) as an effect to supplement the input-output mapping. Symbolic summarization is crucial for the analysis to scale with the number of source APIs of interest.

The second contribution is a set of techniques aimed at efficient propagation of abstract states during the computation. These techniques include escape analysis [6, 7, 44], access-based localization [28], and bypassing [27]. The idea behind these techniques are not new. Our contribution is applying these techniques to optimize an IFDS-style taint analysis, which is an opportunity overlooked by the previous approaches [3, 42]. We identify three types of unnecessary propagation that frequently happen while analyzing taint flows in Android apps, and select optimization techniques that can be seamlessly integrated into an IFDS-style taint analysis. In case of escape-analysis, we report a simple but novel analysis technique that is specifically designed to detect temporary local objects, such as intermediate string values and *StringBuilder* objects. We find that a large portion of temporary objects are local to their allocation sites; therefore, the analysis does not need to propagate information about such objects beyond the allocation sites.

The third contribution is our large-scale evaluation of taint analysis on $400,000$ applications. We note that the precision of IFDS-based taint analysis approaches is well studied (*e.g.*, [3]), and our focus in this paper is on performance and scalability. The dataset we used for this purpose is one of the largest reported in academic literature, and we believe it sheds light on the efficacy and practicality of taint analysis under tight production constraints, and can help guide further research in this area.

2. Overview

2.1. Preliminaries

2.1.1. Taint Analysis

Taint analysis tracks flow of information through a program. The goal is to determine whether the result of a sensitive source API call (*e.g.*, accessing user's contacts) can influence an argument passed to a sink API (*e.g.*, a network write). Any such influence might allow adversaries to infer users' sensitive data based on observations of the sink API arguments. In this work, we ignore information leaks due to control dependence — a common assumption used in practice. Under that assumption, the relevant semantics are that a value is considered tainted if: (1) it is the result of a sensitive API, or (2) it is computed using a tainted value.

Consider the following simple example program, where the sensitive values computed by expressions α and β are assigned to variable `x`. Expressions α and β correspond to sensitive API calls and are called taint sources. The expression `SINK` corresponds to sensitive API calls that reveal information to the adversary. The question mark in the second line denotes an arbitrary conditional expression:

```
1 :  int x=α;
2 :  if(?)  x=β;
3 :  SINK(x);
```

One way to implement a static taint analysis is to use taint facts to track a tainted predicate for all program state (variables on the stack, objects on the heap, globals). A taint fact is a tuple $\langle x, \alpha \rangle$ indicating a memory location x

is tainted by a source API α. For the given program, the taint fact $\langle x, \alpha \rangle$ will be generated for line `1`, which means that variable x may contain information from source α. Similarly, the taint fact $\langle x, \beta \rangle$ will be generated for line `2`. A flow-sensitive analysis would merge the information flowing from line `1` and line `2`, and the taint fact set $\{\langle x, \alpha \rangle, \langle x, \beta \rangle\}$ will be propagated to line `3`. The static analyzer can therefore infer that information from sources α and β may leak to `SINK`.

2.1.2. IFDS Framework and RHS Algorithm

In the taint analysis approach as framed above, we assume that the points-to analysis is performed before the taint analysis, and the taint facts can refer to abstract heap locations. This assumption has two important consequences. First, the abstract domain (set of taint facts) is a powerset domain. Second, the abstract transfer function for taint propagation can be easily seen to be distributive.[2] The category of static analysis problems satisfying the above two properties is referred to as the *IFDS* [34] framework.

For problems that fall in the IFDS framework, there exists an efficient algorithm, the Reps-Horowitz-Sagiv tabulation solver [34] or *RHS*[3], to obtain a fully flow-and-context-sensitive solution. It leverages the distributivity of the transfer function over the powerset domain $P = 2^D$ to generate summaries that map from elements of the domain D to P. For instance, in the case of taint analysis, the set D includes all possible taint facts $\langle x, \alpha \rangle$ and thus $|D|$ is upper-bounded by the product of the number of abstract memory locations and taint sources.

The number of summary entries is linear in $|D|$, and the RHS algorithm has a complexity of $O(E|D|^3)$ where E is the number of edges in the super graph [34], which is obtained by adding inter-procedural call and return edges to the control flow graphs for each method.

Let us consider a simple program to explain the intuition behind the RHS algorithm:

```
x.foo(α+β);   x.foo(α+γ);
```

The example program involves multiple invocations to the same method. Here α, β, and γ are distinct taint sources (*i.e.*, expressions computing sensitive information). Let y be the formal parameter of method `foo`. A taint analysis will propagate two taint facts $\langle y, \alpha \rangle$ and $\langle y, \beta \rangle$ to the body of method `foo` at the first callsite. Let T_α and T_β be the set of output taint facts for `foo` respectively for each input taint fact α, β. The RHS algorithm creates point-wise summaries for method `foo` as follows:

$$\mathit{summary}(\texttt{foo}) = \quad [\langle y, \alpha \rangle \mapsto T_\alpha,\ \langle y, \beta \rangle \mapsto T_\beta]$$

At the second callsite, method `foo` has to be re-analyzed with a different set of input taint facts ($\langle y, \alpha \rangle$ and $\langle y, \gamma \rangle$). Although this input combination is new,

[2] Given abstract domain D, abstract transfer function $F : D \to D$ is distributive *iff* $\forall a, b \in D. F(a \sqcup b) = F(a) \sqcup F(b)$.

[3] As far as we know, the name RHS has first used by Ball *et al.* [5].

the RHS algorithm can apply a part of the above summary ($\langle y, \alpha \rangle \mapsto T_\alpha$), but has to re-analyze `foo` with the new taint fact $\langle y, \gamma \rangle$.

2.2. Symbolic Summarization with Effect

We now point out a specific type of redundant computation performed by the RHS algorithm when applied to taint analysis, and then present a new technique — symbolic summarization with effect — to address this issue.

Motivating example. The RHS algorithm performs redundant computation when a method has to be analyzed with similar yet different inputs. Let us reconsider the example with method `foo` in the previous section. In the example, for similar input taint facts $\langle y, \alpha \rangle$, $\langle y, \beta \rangle$, and $\langle y, \gamma \rangle$, the taint analysis behaves identically since the transformers either (1) simply propagate the input taint, or (2) generate taints depending on source APIs in the method. In the first case, the only difference is the taints being propagated (α, β, or γ). In the second case, the generated taint does not depend on the input taint fact at all. Thus, it suffices to analyze the body of method `foo` only once for the first input taint fact ($\langle y, \alpha \rangle$) and reuse the result for the rest. Our new technique is based on this observation.

Symbolic reasoning. The desired effect is achieved by introducing the notion of a symbolic taint source. When analyzing method `foo` for the first time, instead of using input taint fact $\langle y, \alpha \rangle$, we use the taint fact $\langle y, \star \rangle$ obtained by replacing the concrete taint source α with a symbolic taint source $\star$. This allows us to generate the following general summary:

$$\mathit{summary}(\texttt{foo}) = [\langle y, \star \rangle \mapsto T_\star]$$

where $T_\star$ is the corresponding analysis result for method `foo`. This generalized summary can be reused for all three input taint facts by substituting the symbolic taint source $\star$ by the corresponding concrete taint source (α, β, or γ). Note that the summary is polymorphic only with respect to the taint source, and does not attempt to generalize across memory locations. Therefore, this technique applies even if the method has multiple parameters.

Effects. The above symbolic summary suffers from a limitation — it conveys the propagation semantics of the taint, but it is not sufficient to deduce the set of sink APIs by which method `foo` can leak tainted data. Since, in the symbolic setting, the method `foo` is analyzed only once with a symbolic taint source, it cannot report the leak from a concrete taint source. We solve this problem by augmenting the summary with a set of sink APIs (or program points) to which the method leaks the sensitive information. The full summary for method `foo` has the following form:

$$\mathit{summary}(\texttt{foo}) = [\langle y, \star \rangle \mapsto (T_\star, R_\star)]$$

Conceptually, $R_\star$ is a set of sink APIs (or program points) to which the source API information is leaked during the execution of the method, when parameter

y is tainted. This can be computed while analyzing the body of the method, and used at the callsites of the method to generate precise warnings in combination with the exact input taints available at the callsites. We refer to $R_\star$ in the summary as *effect*, analogous to the effect in the type-and-effect system [22].

Relation to IDE. We now explain the relation of the symbolic summarization technique to the *interprocedural distributive environment-transformer framework (IDE framework)* [36]. It is possible to implement the taint analysis via the IDE framework by treating sink expressions as virtual variables and source expressions as values. However, this would still be inefficient because the tabulation solver for IDE has $O(E|S|^2|L|^3)$ time complexity for our problem, where S is the set of source expressions and L is the set of abstract memory locations (variables + abstract objects).[4] Although this is an improvement over the complexity of a naïve IFDS-style implementation, which is $O(E|S|^3|L|^3)$,[5] symbolic summarization provides an even more significant improvement. The effect of symbolic summarization is to make $|S|$ very small, effectively decreasing the complexity of the analysis to $O(E|L|^3)$. While *STAR* has better worst-case complexity, the question on how the empirical complexity of the two approaches compares remains open and is an opportunity for future work.

Other applications. The symbolic summarization technique can be applied to any IFDS problem meeting the following additional requirements:

- The abstract domain is a Cartesian product of multiple domains, *i.e.*, $D = D_1 \times \cdots \times D_n$.
- There exists a subdomain D_i of D, such that all abstract transfer functions (f) satisfy the following condition: $\forall a, b \in D\ .\ \forall x, y \in D_i\ .\ \Big(\big(a = (b[x/y])\big) \implies \big(f(a) = (f(b))[x/y]\big)\Big)$, where $b[x/y]$ denotes the result of replacing all occurrence of x in b with y.

If the above conditions hold, the analysis can freely replace the i-th component of an abstract state with a symbolic abstract value. Taint analysis and dependency analysis are typical problems satisfying these requirements.

[4] The original IDE paper [36, p. 151] states that the time complexity of the IDE solver is $O(E|L|^3)$, where *the constant of proportionality depends on the height of the micro transfer function domain.* The micro transfer function domain for our problem is determined by the input program, unlike program-independent examples given in the original IDE paper. In the multi-source taint propagation problem, micro transfer functions are elements of a map domain $S \to S$, and the height of the domain is $|S|^2$, where S is the set of source instructions. Taking this into account, the time complexity of the IDE solver for our problem becomes $O(E|S|^2|L|^3)$.

[5] The time complexity of the RHS algorithm for the IFDS framework is $O(E|D|^3)$ where $|D|$ is the size of the abstract domain. For the taint analysis, $|D| = |S||L|$.

Syntactic Components

$x, y, z, ret \in Var \quad l \in Label \quad f \in Field \quad \alpha, \beta, \gamma \in Src$

$Stmt \quad s ::= l : c \mid s_1; s_2 \mid \texttt{choose}(s_1, s_2)$

$Cmd \quad c ::= x = y \mid x = y.f \mid x.f = y \mid \texttt{return}\ x \mid x = y.f(z) \mid x = \alpha \mid \texttt{sink}(x)$

Abstract Program

$program ::= (s, pointsTo, callee)$	$\ell \in \hat{Loc} = \hat{Obj} \times Field + Var$
$pointsTo \in Label \times \hat{Loc} \to 2^{\hat{Obj}}$	$o \in \hat{Obj}$
$callee \in Label \to MID \times Var \times Stmt$	$m \in MID \quad (method\ id)$

Figure 1: Model language definition.

2.3. State Pruning

While eliminating some redundancy, symbolic summarization cannot eliminate all unnecessary information propagation inherent in the analysis. Consider the example:

```
1 :  int x = α;
     // hundreds of lines without using x
n :  SINK(x);
```

A naïve flow-sensitive analysis implementation would generate the taint fact $\langle x, \alpha \rangle$ at line `1`, then propagate the taint fact to line `n` through the irrelevant code block in between. Inefficiency is introduced by the unnecessary propagation of the taint fact through the irrelevant code block. *STAR* mitigates unnecessary propagation via three light-weighted state-pruning techniques: access-based localization [28], bypassing [27], and escape-analysis [7, 44, 6].[6]

The ideas of state-pruning techniques are not new. Yet, the opportunity of applying them to an IFDS-style taint analysis has been overlooked. We find that even an IFDS-style analysis performs a non-trivial amount of unnecessary propagation, and state-pruning techniques are still useful to avoid such wastage. With regards to state-pruning, our contribution is to (1) identify a combination of techniques that work well together for our problem, and (2) propose a new escape-analysis tuned for Android taint analysis. Section 6 explains details of pruning techniques that have been implemented in *STAR*.

3. Taint Analysis: The Baseline

In this section, we formally define the baseline taint analysis, which we gradually improve on in the following sections.

3.1. Model Language

For the formal description of our analyses, we use a loop-free and recursion-free object-oriented model language shown in Fig. 1. Since the abstract domain for taint analysis is finite, loops and recursions do not pose a challenge. The

[6] Note that sparse-analysis [29, 23], a state-of-the-art state-pruning technique, is not applicable because the cost of heap-SSA construction, a pre-processing step required to perform sparse-analysis, is simply too expensive computationally.

language is specialized to simplify exposition of a staged static analysis relying on a pre-analysis for computing call graph and points-to information.

Program statements (s) are uniquely labeled commands ($l : c$), sequences ($s_1; s_2$), and branches ($\texttt{choose}(s_1, s_2)$). Commands include reads/writes to local variables and object fields, method returns/calls, APIs that reveal sensitive information, and APIs that can potentially leak information.

In the model language, a program is composed of three components: the body of the entry point method (s), points-to oracle *pointsTo*, and call-graph oracle *callee*. The class of taint analysis we consider depends on pre-computed call-graph oracle and points-to oracle to resolve function calls and aliasing. The oracles make class definitions and object creation statements (`new`) irrelevant for our analysis.

Call-graph oracle. $\mathit{callee}(l) = (m, x, s)$ denotes that call-graph oracle *callee* has determined that method with ID m is invoked at the callsite labeled l, and the method has formal parameter x and body statement s. For simplicity of presentation, we assume a context-insensitive call-graph construction algorithm (*e.g.*, VTA [39]) and that each callsite has exactly one callee.

Points-to oracle. We denote the set of abstract heap objects computed by the points-to analysis as $\hat{Obj}$. At the program point l, $\mathit{pointsTo}(l, \ell)$ returns the set of abstract objects that may be pointed to by abstract location ℓ. An abstract location ℓ is either a variable (x) or a tuple of abstract object and field ($o.f$), where o is an abstract object representing a group of concrete objects. Note that our memory model is different from purely access-path based model [3, 42]. In our memory model, aliasing between access-paths is implicitly handled by abstract objects; If two access-paths are pointing to the same abstract object, they are aliased. In contrast, in a purely access-path based memory model, aliasing of access-paths needs to be tracked explicitly. For simplicity of presentation, we assume that abstract objects are disjoint, *i.e.*, each concrete object is mapped to exactly one abstract object, and the object abstraction does not change during the analysis [7]. Similarly, we assume a context-insensitive points-to-analysis (*e.g.*, Steensgaard's [38]).

Multiple dispatch and multiple parameters. For simplicity, we restrict our presentation to only consider a single dispatch and a single parameter. At the end of the following section, we explain how the analysis can be extended to remove these restrictions. Our implementation does handle these cases as well.

Tainted references. Though our semantics below assumes that only memory locations of primitive type can have taint labels, our implementation associates taint labels with memory locations of object reference type as well (for more details, see Section 7.2). The semantics that we present can be extended to this case as well by simple extensions to handle accesses to such tainted references.

[7] With a flexible object abstraction (*e.g.*, recency abstraction [4]), object manipulating statements cannot be ignored because they may generate or kill taint facts.

Semantic Components

TaintFact	t	$::= \langle \ell, \alpha \rangle \mid \Lambda$
TaintFactSet	$T \in$	$2^{\mathit{TaintFact}}$
TaintReport	$R \in$	$2^{\mathit{Label} \times \mathit{Src}}$

Abstract Transformer

[Assign]
$$\frac{T' = (T|^{-}_{\{x\}}) \cup \{\langle x, \alpha \rangle \mid \alpha \in T(y)\}}{T \vdash l : x = y \to T', \emptyset}$$

[Load]
$$\frac{T' = (T|^{-}_{\{x\}}) \cup \{\langle x, \alpha \rangle \mid o \in \mathit{pointsTo}(l, y), \alpha \in T(o.f)\}}{T \vdash l : x = y.f \to T', \emptyset}$$

[Store]
$$\frac{T' = T \cup \{\langle o.f, \alpha \rangle \mid \alpha \in T(y), o \in \mathit{pointsTo}(l, x)\}}{T \vdash l : x.f = y \to T', \emptyset}$$

[Taint]
$$\frac{\Lambda \in T \qquad T' = (T|^{-}_{\{x\}}) \cup \{\langle x, \alpha \rangle\}}{T \vdash l : x = \alpha \to T', \emptyset}$$

[Sink]
$$\frac{R = \{(l, \alpha) \mid \alpha \in T(x)\}}{T \vdash l : \texttt{sink}(x) \to T, R}$$

[Return]
$$T \vdash l : \texttt{return}\ x \to T \cup \{\langle ret, \alpha \rangle \mid \alpha \in T(x)\}$$

[Choose]
$$\frac{T \vdash s_1 \to T_1, R_1 \qquad T \vdash s_2 \to T_2, R_2}{T \vdash \texttt{choose}(s_1, s_2) \to T_1 \cup T_2, R_1 \cup R_2}$$

[Seq]
$$\frac{T \vdash s_1 \to T_1, R_1 \qquad T_1 \vdash s_2 \to T_2, R_2}{T \vdash s_1; s_2 \to T_2, R_1 \cup R_2}$$

[Invoke]
$$\frac{\begin{array}{c} \mathit{callee}(l) = (m, x_m, s_m) \\ T_{in} = (T[z/x_m])|^{-}_{Var \setminus \{x_m\}} \qquad T_{in} \vdash s_m \to T_{out}, R \qquad T_{ret} = (T_{out}[ret/x])|^{-}_{Var \setminus \{x\}} \end{array}}{T \vdash l : x = y.f(z) \to (T|^{-}_{\{x\}}) \cup T_{ret}, R}$$

Figure 2: The baseline analysis specification.

3.2. The Analysis Definition

We now provide the definition of a baseline analysis implementing fully flow-and-context-sensitive taint analysis. The analysis serves as a starting point to derive a more scalable analysis. Fig. 2 provides the formal definition of the semantic components and the abstract semantics of the analysis.

Domain. A taint fact t is either a tuple consisting of an abstract location and a taint source ($\langle \ell, \alpha \rangle$), or the sentinel ($\Lambda$). Taint fact $\langle \ell, \alpha \rangle$ denotes that abstract location ℓ (of primitive type) may contain information tainted by sensitive API α. The sentinel (Λ) represents the fact that the relevant program point is reachable. The analysis uses this to avoid analyzing unreachable expressions. T denotes a set of taint facts. We use $T(\ell)$ to denote the set of taint sources that may affect location ℓ (*i.e.*, $T(\ell) = \{\alpha \mid \langle \ell, \alpha \rangle \in T\}$). We define restriction and substitution operations as follows:

$$\begin{array}{lcl} T|_X & \overset{def}{=} & \{\langle \ell, \alpha \rangle \mid \alpha \in T(\ell), \ell \in X\} \cup (T \cap \{\Lambda\}) \\ T|_X^- & \overset{def}{=} & \{\langle \ell, \alpha \rangle \mid \alpha \in T(\ell), \ell \notin X\} \cup (T \cap \{\Lambda\}) \\ T[\ell_1/\ell_2] & \overset{def}{=} & (T|_{\{\ell_1\}}^-) \cup \{\langle \ell_2, \alpha \rangle \mid \alpha \in T(\ell_1)\} \end{array}$$

A taint analysis generates a taint report R, which is a set of pairs consisting of an instruction label and a source. Each element (l, α) of a taint report encodes that information from source α may leak to the sink instruction with label l.

Abstract transformer. The abstract transformer for statements is defined by the judgment $T \vdash s \to T', R$. It denotes that statement s transfers set of input taint facts T to set of output taint facts T' and generates report R.

Assignment, load, store, and source API statements propagate taints from the right-hand side location to the left-hand side location. If the left-hand side is a variable (assignment, load, taint, and invoke statements), the analysis performs a strong update. Note that these statements apply to primitive values. Our analysis depends on the points-to oracle to reason about objects. Also note that the [**Taint**] rule checks whether the set of input taint facts includes the sentinel, avoiding analysis of unreachable source expressions.

Sink statements generate a taint report using the taints associated with the parameter. Return statements propagate taints of the return parameter to the special placeholder variable *ret*. We assume that variable *ret* does not appear in any program. Handling of branch and sequence statements is standard. For branch statements, each branch body is analyzed separately, and the results are merged. For sequence statements, the judgments are applied in sequence.

Handling of the invoke statements involves several projections. The set of input taint facts at the callsite is obtained by substituting the argument of the caller for the callee's formal parameter and then filtering out taint facts related to the caller's local variables. The body of the callee method is analyzed using the projected set of input taint facts. The analysis result of the body of the callee is projected back to the callsite, by removing taint facts about the callee's local variables and by substituting place holder variable *ret* with the actual return variable of the callsite.

Soundness. A taint analysis is sound if and only if it captures all information leaks across all potential concrete executions. In this paper, we focus on proving that the symbolic summarization based analysis is preserving the soundness of the baseline analysis. The formal definitions of information leak, capturing, and soundness, as well as the proof of the soundness preservation theorem are available in Section 5.3.

Supporting multiple dispatch and multiple parameters. Now we explain how to support multiple dispatch and multiple parameters. With multiple dispatch and multiple parameters, *callee* may return a set of tuples, and each tuple may contain a set of formal parameters. Both can be supported by modifying the [**Invoke**] rule. Multiple dispatch can be handled by first analyzing each tuple separately, then merging the resulting taint fact sets and reports to form the final result. Multiple parameters can be supported by modifying the way T_{in} is computed. Assume that a method having a sequence of parameters $(x_m^1, \ldots, x_m^n)$ is invoked with a sequence of arguments $(z_1, \ldots, z_n)$ where each z_i is a variable visible at the callsite. Then, $T_{in} = (T[z_1/x_m^1] \ldots [z_n/x_m^n])|^{-}_{Var \setminus \{x_m^1, \ldots, x_m^n\}}$.

4. IFDS Formulation

In this section, we lift the baseline analysis by leveraging the IFDS framework. In practice, we found that the performance of the lifted analysis (vanilla IFDS-style analysis) is not satisfactory, which led us to develop various optimizations introduced in the subsequent sections.

4.1. The Analysis Definition

Fig. 3 provides a reformulation of the baseline abstract semantics in terms of the IFDS framework.

Pointwise summary. The new formulation uses summaries ($\$$). A summary is a map from a pair of a method identifier and an input taint fact (at method entry) to a set of taint facts (or bottom) at method exit. It encodes how methods propagate taints through their execution. Bottom ($\perp$) indicates that the mapping is undefined for the corresponding key (*i.e.*, the summary is not available for that particular input). The empty summary (denoted $[]$) maps every method identifier and taint fact to $\perp$ (*i.e.*, $\forall m, t.[](m, t) = \perp$). Function summaries form an abstract domain where the empty summary is bottom. The join operator $\sqcup$ and order operator $\sqsubseteq$ are point-wise extensions of the corresponding operations over sets:

$$\begin{aligned} \$_1 \sqcup \$_2(m, t) &\overset{def}{=} \$_1(m, t) \cup \$_2(m, t) \\ \$_1 \sqsubseteq \$_2 &\overset{def}{=} \forall m, t.\$_1(m, t) \subseteq \$_2(m, t) \end{aligned}$$

Semantic Components

Summary $\$ \in \mathit{MID} \times \mathit{TaintFact} \to 2^{\mathit{TaintFact}} \cup \{\bot\}$

Abstract Transformer

[Cmd]
$$\frac{c \not\equiv x = y.f(z) \qquad T \vdash l : c \to T', R}{\$, T \vdash l : c \to \$, T', R}$$

[Seq]
$$\frac{\$, T \vdash s_1 \to \$_1, T_1, R_1 \qquad \$_1, T_1 \vdash s_2 \to \$_2, T_2, R_2}{\$, T \vdash s_1; s_2 \to \$_2, T_2, R_1 \cup R_2}$$

[Choose]
$$\frac{\$, T \vdash s_1 \to \$_1, T_1, R_1 \qquad \$, T \vdash s_2 \to \$_2, T_2, R_2}{\$, T \vdash \mathit{choose}(s_1, s_2) \to \$_1 \sqcup \$_2, T_1 \cup T_2, R_1 \cup R_2}$$

[Invoke]
$$\frac{\begin{array}{c} \mathit{callee}(l) = (m, x_m, s_m) \\ T_{in} = (T[z/x_m])|^{-}_{\mathit{Var}\setminus\{x_m\}} \qquad \forall t_i \in T_{in}. \$, t_i \vdash^* m : s_m \to \$_i, T_i, R_i \\ T_{out} = \cup T_i \qquad T' = (T|^{-}_{\{x\}}) \cup (T_{out}[ret/x]) \qquad \$' = \sqcup \$_i \qquad R' = \cup R_i \end{array}}{\$, T \vdash l : x = y.f(z) \to \$', T', R'}$$

Summary Lookup

[Hit]
$$\frac{\$(m, t) = T}{\$, t \vdash^* m : s \to \$, T, \emptyset}$$

[Miss]
$$\frac{\begin{array}{c} \$(m, t) = \bot \\ \$, \{t\} \vdash s \to \$', T', R \qquad T_{ret} = T'|^{-}_{\mathit{Var}\setminus\{ret\}} \qquad \$'' = \$' \sqcup [(m, t) \mapsto T_{ret}] \end{array}}{\$, t \vdash^* m : s \to \$'', T_{ret}, R}$$

Figure 3: The IFDS-style analysis specification. The highlighted portion denotes the summary lookup judgement.

Semantic Components

$\$ \in \mathit{MID} \times \mathit{TaintFact} \to (2^{\mathit{TaintFact}} \times \mathit{TaintReport}) \cup \{\bot\}$

Summary Lookup

[Miss$_\Lambda$]
$$\frac{\begin{array}{c} \$(m, \Lambda) = \bot \\ \$, \{\Lambda\} \vdash s \to \$', T', R \qquad T_{ret} = T'|^{-}_{\mathit{Var}\setminus\{ret\}} \qquad \$'' = \$' \sqcup [(m, \Lambda) \mapsto (T_{ret}, \emptyset)] \end{array}}{\$, \Lambda \vdash^* m : s \to \$'', T_{ret}, R}$$

[Miss$_\star$]
$$\frac{\begin{array}{c} \$(m, \langle \ell, \star \rangle) = \bot \\ \$, \{\langle \ell, \star \rangle\} \vdash s \to \$', T', R \qquad T_{ret} = T'|^{-}_{\mathit{Var}\setminus\{ret\}} \qquad \$'' = \$' \sqcup [(m, \langle \ell, \star \rangle) \mapsto (T_{ret}, R)] \\ T_{inst} = \{\langle \ell, \alpha \rangle \mid \langle \ell, \star \rangle \in T_{ret}\} \qquad R_{inst} = \{(l, \alpha) \mid (l, \star) \in R\} \end{array}}{\$, \langle \ell, \alpha \rangle \vdash^* m : s \to \$'', T_{inst}, R_{inst}}$$

[Hit$_\Lambda$]
$$\frac{\$(m, \Lambda) = (T, \emptyset)}{\$, \Lambda \vdash^* m : s \to \$, T, \emptyset}$$

[Hit$_\star$]
$$\frac{\$(m, \langle \ell, \star \rangle) = (T, R) \qquad T_{inst} = \{\langle \ell, \alpha \rangle \mid \langle \ell, \star \rangle \in T\} \qquad R_{inst} = \{(l, \alpha) \mid (l, \star) \in R\}}{\$, \langle \ell, \alpha \rangle \vdash^* m : s \to \$, T_{inst}, R_{inst}}$$

Figure 4: Symbolic summarization specification. Note that taint facts may now also contain symbolic taint ($\star$).

Abstract transformer. The abstract transformer judgment $\$, T \vdash s \rightarrow \$', T', R$ denotes that statement s transforms input summary $\$$ and input taint set T to output summary $\$'$ and output taint set T', and generates taint report R. The transfer function is a straightforward extension of the transfer functions in Fig. 2 (handled by the **[Cmd]** rule), with a few differences. First, the **[Seq]** and **[Choose]** rules are extended to update the summary. Second, when handling a method call, **[Invoke]** treats each taint fact in the set of incoming taint facts point-wise (highlighted line in Fig. 3). The handling of each input taint fact is delegated to the summary lookup judgment.

The summary lookup judgment $\$, t \vdash^* m : s \rightarrow \$', T, R$ means that statement body s of method m receives input taint fact t and input summary $\$$, and generates updated summary $\$'$, output taint set T, and report R. The judgment has two cases: If the input taint fact was observed before, the entry value is reused (**[Hit]**). The method body is analyzed only if the input taint fact is not observed yet (**[Miss]**).

5. Symbolic Summarization

This section describes the symbolic summarization technique, which aims to remove redundant computation performed by the IFDS-style analysis. We start with a formal definition, and then explain the idea with an example. The resulting analysis is sound w.r.t. the baseline analysis.

5.1. The Analysis Definition

Symbolic summarization. Fig. 4 defines symbolic point-wise summary-based taint analysis. The summary is now redefined to have a taint report as an effect. A taint report is a set of sink APIs. Similar to the domain for IFDS-style analysis, the set of summaries constitutes an abstract domain. The join and order operators are point-wise extensions.

Abstract transformer. The new transfer function differs from IFDS-style semantics only with respect to function summaries. The summary lookup judgment has four cases. **[Hit$_\star$]** and **[Miss$_\star$]** rules handle summary lookup when the input taint fact is a usual taint fact (*i.e.*, not Λ). Intuitively, these rules cover the input-dependent behavior of the target method. Both cases use a symbolic input taint fact ($\langle \ell, \star \rangle$) to lookup the method summary. If the lookup succeeds (**[Hit$_\star$]**), the resulting set of output taint facts and the taint report are instantiated and returned. The instantiation process substitutes the symbolic taint source $\star$ with the actual taint source α at the callsite. If the lookup fails (**[Miss$_\star$]**), the body of the target method is analyzed with the symbolic input taint fact. The summary is updated with the resulting output taint fact and taint report without instantiation, while the instantiated version is returned to the callsite.

Other cases are handled by **[Hit$_\Lambda$]** and **[Miss$_\Lambda$]** rules. These rules cover the input-independent behavior of the target method. If the lookup succeeds, the

```
static int foo(int w){
  SINK(w);
  SINK(γ);
  return w + γ;
}
static void bar(){
  int x, y;
  x = foo(α);
  y = foo(β);
}
```

Summary of `foo` (RHS style)

input facts	output facts
$\langle w, \alpha \rangle$	$\{\langle ret, \alpha \rangle\}$
$\langle w, \beta \rangle$	$\{\langle ret, \beta \rangle\}$
Λ	$\{\langle ret, \gamma \rangle, \Lambda\}$

Summary of `foo` (symbolic)

input facts	output facts	effect
$\langle w, \star \rangle$	$\{\langle ret, \star \rangle\}$	$\{(2, \star)\}$
Λ	$\{\langle ret, \gamma \rangle, \Lambda\}$	$\{\ \}$

Figure 5: Example Java program composed of two static methods from a single class. α, β, and γ are source expressions of `int` type computing sensitive information. `SINK` leaks sensitive information.

lookup results are directly returned to the callsite. If it fails, the body of the method m is analyzed, the summary is updated, and the result is returned to the callsite. Note that the taint report obtained by analyzing the method body is not added to the method summary. When the sentinel fact is the only input taint fact for the method, the resulting taint report cannot have a symbolic entry.[8] This implies that the taint report need not be added to the summary.

Properties. The symbolic summarization based analysis is sound and complete w.r.t. the baseline analysis, *i.e.*, they compute the same taint report. The proof is in Section 5.3. The analysis uses finite domains, and the transformer is monotonic. Hence, the analysis always terminates. The monotonicity of the transformer can be proved by induction on the height of the derivation tree.

5.2. Example Run of the Analysis

We use the program in Fig. 5 to illustrate how the symbolic summarization works. The example program has two methods `foo` and `bar`. The `foo` method leaks the input parameter w and the local source API γ to a `SINK`. Then it returns the sum of `w` and γ. Note that γ is a source API. Method `bar` is the entry point of the program, and it invokes method `foo` twice with different inputs. The return values are assigned to local variable `x` and `y`. During the execution of the program, the information of source APIs α and β leak to `SINK` at line 2. Let us follow the analysis step by step.

Step 1: line 7. The analysis of the entry point starts with a singleton set containing the sentinel taint fact $\{\Lambda\}$. Line 7 merely propagates $\{\Lambda\}$ to the next line.

[8] To be more precise, if the input taint fact is concrete (*i.e.*, it is the sentinel or a taint fact with concrete taint source), the analysis yields a concrete output taint fact set and a concrete taint report as outputs. Similarly, if the input taint fact is symbolic, the analysis yields a symbolic taint fact set and a symbolic taint report. See Section 5.3 for more details.

Step 2: line 8 calling `foo` with α. This statement receives set of input taint facts $\{\Lambda\}$, and invokes method `foo` with argument α containing sensitive information. The analysis first computes the set of input taint facts for `foo` as $\{\langle w, \alpha\rangle,\ \Lambda\}$, and then looks up the summary of `foo` for each input taint fact. The lookup for $\langle w, \alpha\rangle$ is generalized by substituting the concrete taint source α with symbolic taint source $\star$. This lookup fails since `foo` has never been analyzed before, and so the analysis descends into `foo`.

Step 3-1: analyzing `foo` with $\langle w, \star\rangle$. Due to the leak at line 2, the analysis adds the symbolic report entry $(2, \star)$ to the report. A new taint fact $\langle ret, \star\rangle$ will be generated at line 4, since the method returns the information contained in variable w. Taint source γ is ignored since propagating γ to the return variable is an input-independent behavior, whereas the analysis is currently handling an input-dependent behavior of `foo`. Such input-independent behaviors will be handled while analyzing the same method with the sentinel input. The analysis concludes that `foo` generates the set of taint facts $\{\langle w, \star\rangle,\ \langle ret, \star\rangle\}$ given the set of taint facts $\langle w, \star\rangle$. The set of taint facts and the report returned to the callsite is therefore $\{\langle ret, \star\rangle\}$ and $\{(2, \star)\}$ The following mapping is also added to the summary: $[(\texttt{foo}, \langle w, \star\rangle) \mapsto (\{\langle ret, \star\rangle\}, \{(2, \star)\})]$.

Step 3-2: analyzing `foo` with Λ. This time, the analysis is covering the input-independent behavior of the method. First, the analysis generates a new taint fact $\langle ret, \gamma\rangle$ at line 4, and then the concrete report entry $(3, \gamma)$ at line 3. At the end, the set of taint facts $\{\langle ret, \gamma\rangle, \Lambda\}$ and the report $\{(3, \gamma)\}$ are returned to the callsite. The summary is updated with new entry $(\texttt{foo}, \Lambda) \mapsto (\{\langle ret, \gamma\rangle, \Lambda\}, \emptyset)$. As the report contains only a concrete entry, the analysis doesn't add the report to the summary.

Step 4: line 8, returning from `foo`. The analysis first instantiates the `foo` summary with different input taints, replacing *ret* with the actual variable x and $\star$ with the actual taint source α. Next, the analysis merges the results of that instantiation with the input taint fact at the callsite, yielding taint facts $\{\langle x, \alpha\rangle,\ \langle x, \gamma\rangle,\ \Lambda\}$. By instantiating the effect part of the summary, the analysis detects that taint source α leaks to SINK at line 2, and adds the corresponding concrete report entry $(2, \alpha)$ to the report. Note that report entry $(3, \gamma)$ is already reported while analyzing method `foo`.

Step 5: line 9. This statement receives input taint set $\{\langle x, \alpha\rangle,\ \langle x, \gamma\rangle,\ \Lambda\}$, and invokes `foo` method with new argument β. Our analysis looks up the summary of `foo` for input taints ($\langle w, \beta\rangle$ and Λ) and resolves inputs using the summary: Λ is trivially resolved, as it was observed before, while $\langle w, \beta\rangle$ is resolved by first generalizing it to $\langle w, \star\rangle$ and then by looking up the summary. The analysis instantiates the lookup results, projects them to the callsite, and then merges them with the incoming taint set and report, yielding taint facts $\{\langle x, \alpha\rangle,\ \langle x, \gamma\rangle,\ \langle y, \beta\rangle,\ \langle y, \gamma\rangle,\ \Lambda\}$ and report $\{(2, \alpha), (2, \beta),\ (3, \gamma)\}$.

5.3. Soundness of Symbolic Summarization

In this section, we prove that the symbolic summarization based analysis and the baseline analysis compute the same set of reports. As a first step to prove this property, we begin with defining *well-formed summaries*, *sound summaries*, *covering reports*, *concrete (abstract) taint fact sets*, and *concrete (abstract) taint reports*.

Definition 1 (Well-formed Summary)**.** Summary $\$$ is well-formed *if and only if* the following holds:

$$\begin{array}{l} \forall m \in \mathit{MID} \;.\; \$(m, \Lambda) \neq \bot \implies \$(m, \Lambda) = (\_, \emptyset) \\ \forall m \in \mathit{MID}, \langle \ell, \alpha \rangle \in \mathit{TaintFact} \;.\; \alpha \neq \star \implies \$(m, \langle \ell, \alpha \rangle) = \bot \end{array}$$

Definition 2 (Sound Summary)**.** Summary $\$$ is sound *if and only if* it is well-formed and the following is true:

$$\begin{array}{l} \forall m \in \mathit{MID} \;.\; \$(m, \Lambda) = (T, \emptyset) \implies \exists R. \{\Lambda\} \vdash s_m \to (T, R) \\ \forall m \in \mathit{MID}, t \in \mathit{TaintFact} \;.\; t \neq \Lambda \wedge \$(m, t) = (T, R) \implies \{t\} \vdash s_m \to (T, R) \end{array}$$

where s_m indicates the body statement of method m. Note that we are defining the soundness of the summary with respect to the abstract semantics of the baseline analysis. This is essential to the soundness proof.

Definition 3 (Covering Report)**.** Let $\$$ be a sound summary. Taint report R covers $\$$ *if and only if* the following is true:

$$\forall m \in \mathit{MID} \;.\; (\$(m, \Lambda) = (T, \emptyset) \wedge \{\Lambda\} \vdash s_m \to T, R') \implies R' \subseteq R.$$

Again, note that we are defining the property with respect to the abstract semantics of the baseline analysis. Also note that the definition only considers the defined entries of the summary. Because of this, the empty report covers the empty summary.

Definition 4 (Concrete Taint Fact Sets and Concrete Reports)**.** We say that a set of taint facts is concrete if it is either the sentinel (Λ) or a taint fact with a concrete taint source (such as α). A taint fact containing a symbolic taint source ($\star$) is symbolic. We say that a set of taint facts is concrete (resp. symbolic) if and only if all the elements are concrete (resp. symbolic). We say that a taint report is concrete (resp. symbolic) if all the entries of the report are composed of a concrete (resp. symbolic) taint sources.

The second step of the soundness proof is to establish that the analysis using a concrete input and the analysis using the corresponding symbolic input followed by an additional instantiation step yield the identical result:

Lemma 1 (Baseline Instantiation)**.** *Let ℓ and ℓ' be abstract locations, α be a taint source, T_1 and T_2 be sets of taint facts, R_1 and R_2 be taint reports, and s be a statement. Assume $\{\langle \ell, \star \rangle\} \vdash s \to T_1, R_1$ and $\{\langle \ell, \alpha \rangle\} \vdash s \to T_2, R_2$. Then, the following is true:*

$$T_2 = \{\langle \ell', \alpha \rangle \mid \langle \ell', \star \rangle \in T_1\} \;\wedge\; R_2 = \{(l, \alpha) \mid (l, \star) \in R_1\}$$

Proof. Proof by the induction on the height of the derivation of the baseline analysis. □

The soundness of the symbolic summarization based analysis follows from the following proposition and the soundness of the baseline analysis. The proposition aims to show two properties: First, the baseline analysis and the symbolic summarization based analysis compute equivalent results. Second, the concreteness of the output taint fact set and the output report depend on the concreteness of the input taint fact set. In other words, the analysis preserves concreteness.

Proposition 2. *Let $\$$ be a sound summary, R be a concrete report covering $\$$, s be a statement, and T, T_1 and T_2 be sets of taint facts. Assume T is either concrete or symbolic, $\$, T \vdash s \to \$_1, T_1, R_1$, and $T \vdash s \to T_2, R_2$. Then, the following is true:*

$$\begin{aligned}&T_1 = T_2 \ \wedge\ R_1 \cup R = R_2 \ \wedge\ \$_1 \textit{ is sound} \ \wedge\ R_1 \cup R \textit{ covers } \$_1\\ &\wedge (T \textit{ is concrete} \implies T_1 \textit{ and } R_1 \textit{ are concrete})\\ &\wedge (T \textit{ is symbolic} \implies T_1 \textit{ and } R_1 \textit{ are symbolic})\end{aligned}$$

Proof. Proof by the induction on the height of the derivation of the symbolic summarization based analysis.

- [**Invoke**] case: We begin by applying the distributivity of the baseline abstract transformer. The case boils down to showing that for every t_i in T, the summary lookup $\$, t_i \vdash^* m : s_m \to \$_1, T_1, R_1$ and the baseline semantics $\{t_i\} \vdash s_m \to T_2, R_2$ computes the same result ($T_1 = T_2$, $R \cup R_1 = R_2$, $\$_1$ is sound, and $R \cup R_1$ covers $\$_1$), and as well as the additional property (if t_i is concrete, then T_1 and R_1 are concrete). The definition of summary lookup requires us to perform a case analysis on the summary lookup rules:
  - The hit cases follow from the soundness of the summary and the induction hypothesis.
  - The miss cases follow from Lemma 1 and the induction hypothesis. Here, the proof uses the fact that the analysis preserves the concreteness to show the equivalence of the results. For example, in the [**Miss**$_\Lambda$] case, the induction hypothesis and the concreteness of the input taint fact implies the concreteness of the output taint report R. Therefore, directly returning the report without putting it into the summary is sufficient to show that the resulting summary $\$''$ is sound, and the resulting report R covers $\$''$. The exact opposite happens in the [**Miss**$_\star$] case — The induction hypothesis and the fact that input taint fact is symbolic implies that the output taint report R is also symbolic. Hence, directly putting it into the summary without returning it is sufficient to show that the resulting summary $\$''$ is sound and the resulting report R_{inst} covers $\$''$.

- Other cases can be easily proved using the soundness of the summary and induction hypothesis.

□

Corollary 3 (Soundness Preserving)**.** *The symbolic summarization based analysis is sound w.r.t. the baseline analysis. I.e., if* $\{\Lambda\} \vdash s \rightarrow T_1, R_1$*, and* $[], \{\Lambda\} \vdash s \rightarrow \$', T_2, R_2$*, then* $R_1 = R_2$*.*

Proof. Let R be a taint report covering the empty summary. By definition, the empty summary is a sound summary. Hence, we can apply Proposition 2, which yields $R \cup R_2 = R_1$. Therefore, we can replace R with the empty report, which yields $R_2 = R_1$. In other words, the summarization based analysis and the baseline analysis always compute the same taint report. □

6. State Pruning

The taint analysis described in the previous section propagates abstract states (taint facts, in our case) over the control flow edges. However, the propagation is not cognizant of whether the propagated abstract state is used by the target program point. Indeed, several recent works [15, 19, 23, 29] report that pruning the abstract state by forgetting irrelevant facts can provide significant performance improvement.

STAR employs three state-pruning techniques: escape-based pruning, access-based localization, and bypassing. Although the ideas of state-pruning techniques are not new, the potential of using these technique has been overlooked in the previously reported IFDS-style taint analyzers [3, 42]. Yet, we find that state-pruning can play an important role in scaling an IFDS-style taint analysis. While implementing escape-based pruning, we also found a simple escape analysis that works effectively and efficiently for our problem. In this section, we explain the basics of each state-pruning technique using the example in Fig. 6. We also explain how each technique can be seamlessly integrated into the IFDS-style taint analysis.

6.1. Escape-Based Pruning

When handling a function call, the return state can be pruned with respect to abstract objects that are not accessible to the caller. An over-approximation of object lifetime [25, 40, 6, 35] can be used to filter the returned abstract state. An instance of this idea is to use an escape analysis [6, 7, 44] to infer whether an object escapes its creation scope.

Consider class O and class A of the example in Fig. 6. Assume that there is an entry point method that creates an instance of A and invokes method `bar`. In this scenario, variable x contains the object created at line 9, whose field f, g, and h are tainted. Then, the content of field f of the object in variable x is returned. A naïve analyzer would conclude that taint facts $\langle o_{\mathtt{x}}.f, \alpha\rangle$, $\langle o_{\mathtt{x}}.g, \beta\rangle$, $\langle o_{\mathtt{x}}.h, \gamma\rangle$, and $\langle ret, \alpha\rangle$ should be returned to the callsite. However, the object in x is local to method `foo`. Hence, it is sufficient to return $\langle ret, \alpha\rangle$, and taint facts about the object need not be returned to the caller `bar`.

```
class O {
  int f=α;
  int g=β;
  int h=γ;
}

class A {
  int foo() {
    O x = new O();
    return x.f;
  }

  void bar() {
    print(foo());
  }
}
```

```
class B {
  void zoo(O x) {
    print(x.f);
  }

  void koo(O y) {
    zoo(y);
  }

  void car() {
    O z = new O();
    print(z.g);
    print(z.h);
    koo(z);
  }
}
```

Figure 6: Example Java program. We assume that there is entry method `main` that creates an instance of A and B, and then invokes methods `bar` and `car`.

Abstract semantics. This optimization affects the $[\mathbf{Miss}_\Lambda]$ rule and the $[\mathbf{Miss}_\star]$ rule for handling the returns from a method call. We assume a pre-analysis, which computes $\neg \mathit{escape}(m)$ ($\subseteq \hat{Loc}$) a sound over-approximation of the set of abstract objects that do not escape method m. The analysis uses the non-escaping information to filter the analysis result of the method body as follows:

$$[\mathbf{Miss}_\Lambda]\ \frac{\$(m,\Lambda)=\bot \qquad \$,\{\Lambda\}\vdash s \to \$',T',R \qquad T_{ret}=(T'|^{-}_{Var\setminus\{ret\}})|^{-}_{\neg escape(m)} \qquad \$''=\$'\sqcup[(m,\Lambda)\mapsto(T_{ret},\emptyset)]}{\$,\Lambda\vdash^{*} m : s \to \$'',T_{ret},R}$$

The $[\mathbf{Miss}_\star]$ rule is modified similarly.

Escape analysis in STAR. STAR performs simple reasoning to determine if an abstract object is local to a method or not. To escape the allocation site, an object needs to be returned, passed to another method as an argument, assigned to an object field, or assigned to a static field of a class; all of these operations create an alias. If an abstract object has only one access-path pointing to it, which can be easily determined by querying an underlying points-to oracle, we can conclude that the abstract object cannot escape its allocation site. Although the approach seems to be simplistic, we found it to be extremely effective at detecting taint facts about temporary local objects, especially string values and *StringBuilder* objects. Section 7.3 provides more explanation regarding the effectiveness of this simple escape analysis.

The necessity of escape-based pruning. A reader familiar with an access-path based taint analysis (*e.g.*, FlowDroid [3]) may think that method local states can be detected without using an escape analysis. That is true when the analysis is using a purely access-path based memory model; in such a case, any taint facts about method local access-paths can be easily identified and removed by simply checking root variables of access-paths. However, if an analysis uses abstract objects to reason about object fields, like *STAR*, the analysis needs a separate mechanism to determine and remove method local objects because this information is not available in the abstract objects themselves. We also emphasize that other two state-pruning techniques are orthogonal to the choice of memory model.

6.2. Access Based Localization

Given an over-approximation of the abstract program state that is accessed by a method, one can partition the abstract state at the callsite. One partition concerns the abstract program state that may be accessed by the callee and the rest concerns program state irrelevant to the callee. It is sufficient to analyze the callee using the former and then obtain the post-state by joining with the later. This idea is known as framing [30]. *STAR* uses access-based localization [28], an instance of framing, which uses context-insensitive mod-ref analysis as a pre-analysis to over-approximate the abstract state that might be accessed by each callee (and its transitive callees).

Consider class O and class B of the example in Fig. 6. Assume that there is an entry point method which creates an instance of B and invokes method `car`. In the example, field f, g, and h are tainted by the constructor of O. Hence, the analysis will generate taint facts $\langle o_{\mathtt{z}}.f, \alpha\rangle$, $\langle o_{\mathtt{z}}.g, \beta\rangle$, and $\langle o_{\mathtt{z}}.h, \gamma\rangle$ at line 29, where $o_{\mathtt{z}}$ denotes the abstract object created at line 29. Among these, only the taint fact about field f is necessary to analyze method `koo` since it is accessed by `koo` and its transitive callee `zoo`.

Abstract semantics. The optimization affects the **[Invoke]** rule. We use $access(m)$ $(\subseteq \hat{Loc})$, a sound over-approximation of the access set of method m, to filter the input taint fact set passed to the callee:

$$\textbf{[Invoke]}\ \frac{\begin{array}{c} callee(l) = (m, x_m, s_m) \\ T_{in} = ((T[z/x_m])|^{-}_{Var\setminus\{x_m\}})|_{access(m)} \qquad \forall t_i \in T_{in}.\$, t_i \vdash^* m : s_m \rightarrow \$_i, T_i, R_i \\ T_{out} = \cup T_i \qquad T' = T|^{-}_{\{x\}} \cup T_{out}[ret/x] \qquad \$' = \sqcup \$_i \qquad R' = \cup R_i \end{array}}{\$, T \vdash l : x = y.f(z) \rightarrow \$', T', R'}$$

6.3. Bypassing

We rely on a bypassing technique [27] to reduce the size of abstract states propagated intra-procedurally. Bypassing is based on the observation that methods typically only access a small portion of the abstract program state, and that the rest is accessed by its transitive callees only. Therefore, the analysis can partition an abstract state at method entry into local state, which may be accessed by the method directly, and non-local state, which is accessed only by its callees. The local state is propagated through the control flow graph, whereas the non-local state is directly forwarded to the set of callsites that are immediate post-dominators of the program point, skipping the local instructions in between.

Consider class O and class B of the example in Fig. 6 again. In the body of method `car`, the field f of the object in variable z is never used directly by method `car`. Bypassing partitions taint facts into local facts ($\langle o_{\mathtt{z}}.g, \beta\rangle$ and $\langle o_{\mathtt{z}}.h, \gamma\rangle$) and non-local facts ($\langle o_{\mathtt{z}}.f, \alpha\rangle$), and short-circuits the non-local facts to the next invoke instruction (line 32), skipping the local instructions (line 30 and 31) in between.

Abstract semantics. We need to modify the abstract semantics judgment to the following form:

$$\$, T_L, T_B, m \vdash s \to \$', T'_L, T'_B, R$$

The new judgment takes a pair of taint fact sets as input (resp. local set T_L and bypassing set T_B), and generates a pair of output taint sets (resp. output local set T'_L and output bypassing set T'_B.) Bypassing sets will be only used and modified by invoke statements. Also note that the judgment is augmented with m, the method enclosing the current statement s. We explain the two rules with interesting modifications (modifications required for other rules are trivial):

$$\textbf{[Invoke]}\ \frac{\begin{array}{c} \mathit{callee}(l) = (m, x_m, s_m) \\ T_{in} = (\ (T_L \cup T_B)\ [z/x_m])|^{-}_{(Var\backslash\{x_m\})} \\ \forall t_i \in T_{in}.\$, t_i \vdash^* m : s_m \to \$_i, T_i, R_i \\ T_{out} = \cup T_i \qquad T' = T|^{-}_{\{x\}} \cup T_{out}[ret/x] \qquad \$' = \sqcup \$_i \qquad R' = \cup R_i \\ T'_L = T'|_{\mathit{localAcc}(m')} \qquad T'_B = T'|^{-}_{(\mathit{localAcc}(m'))} \end{array}}{\$, T_L, T_B, m' \vdash l : x = y.f(z) \to \$', T'_L, T'_B, R'}$$

The modified **[Invoke]** rule, where method dispatching occurs, first merges incoming bypassing taint facts and local taint facts, and then passes the merged result to the callee. After analyzing the callee, the returned taint fact set (T') is split again using the method local access information ($\mathit{localAcc}(m) \subseteq \hat{Loc}$.)

The rules handling summary lookup-failures are a counter part to the **[Invoke]** rule, and they are modified accordingly:

$$[\mathbf{Miss}^{L}_{\star}]\ \frac{\begin{array}{c} \$(m, \langle \ell, \star\rangle) = \bot \qquad \ell \in \mathit{localAcc}(m) \\ \$, \{\langle \ell, \star\rangle\}, \{\}, m \vdash s \to \$', T_L, T_B, R \qquad T_{ret} = \ (T_L \cup T_B)\ |^{-}_{Var\backslash\{ret\}} \\ T_{inst} = \{\langle \ell, \alpha\rangle \mid \langle \ell, \star\rangle \in T_{ret}\} \\ \$'' = \$' \sqcup [(m, \langle \ell, \star\rangle) \mapsto (T_{ret}, R)] \qquad R_{inst} = \{(l, \alpha) \mid (l, \star) \in R\} \end{array}}{\$, \langle \ell, \alpha\rangle \vdash^* m : s \to \$'', T_{inst}, R_{inst}}$$

The $[\mathbf{Miss}_{\star}]$ rule is now split into two cases. The $[\mathbf{Miss}^{L}_{\star}]$ rule (L stands for *local*) handles the case when the input taint fact is about a memory location that is accessed by the callee. If this is the case, the symbolic version of the taint fact will be added to the input local taint fact set, as usual. The analysis result for the method body (s) is merged and then returned. Note that the sequence of actions (split-then-merge) mirrors what happens in the **[Index]** rule (merge-then-split). The other case (the $[\mathbf{Miss}^{B}_{\star}]$ rule where B stands for *bypassing*) is similarly defined: if the input taint fact is not about a locally accessed memory location, the input taint fact is added to the bypassing set.

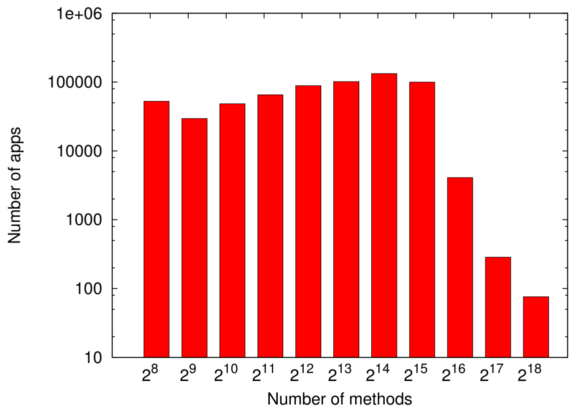


Figure 7: Histogram showing the number of methods per app in the dataset.

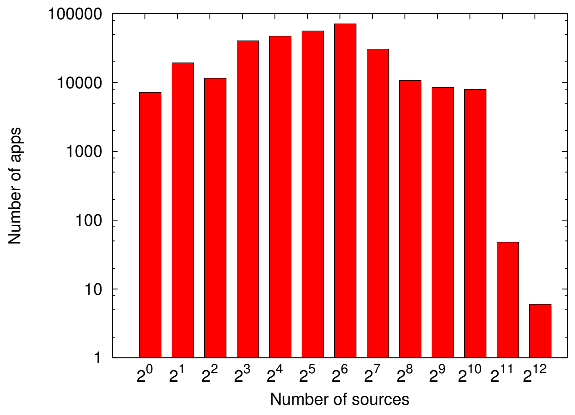


Figure 8: Histogram showing the number of sources per app in the dataset.

7. Evaluation

We aim to answer the following research questions through the evaluation: (1) How does each optimization technique affect the analysis time and the memory consumption? (2) How do symbolic summarization and escape-based pruning compare in their effectiveness to other optimization techniques? (3) What is the overall performance of *STAR* with all optimizations in comparison to basic IFDS-style analysis?

We focus on the problem of improving the performance of an IFDS-style analysis. Regarding precision, we confirmed that all optimizations preserve the result of the baseline analysis unchanged (*i.e.*, the same set of reports are generated). The evaluation of the precision of the baseline analysis is out of scope of this paper.

7.1. Evaluation Corpus

The benchmark used for evaluation is a collection of 400,000 Android apps, all of which call at least one source API. Fig. 7 shows the distribution of the number of methods per app, which we use as a metric to measure the size of apps (other metrics such as number of instructions/classes show a similar distribution). While counting the methods, we first constructed a call-graph and ignored methods that are not reachable from one of entry-point methods. Entry-point methods will be explained in Section 7.2. The largest app in our dataset has over 200,000 methods. 25% of apps have fewer than 10,000 methods, and 50% of apps have fewer than 25,000 methods.

7.2. Implementation

We implemented *STAR* on top of our Android/Java static analysis framework implemented in C++. The frontend accepts Dex/Java bytecode allowing the framework to work directly on APKs (the package file format for Android apps) or JARs (the package file format for Java class files), without requiring source code access. The framework features a rich set of infrastructure passes (*e.g.*, SSA[9], dominance computation, type inference, escape analysis,...), which simplifies developing more complicated client passes.

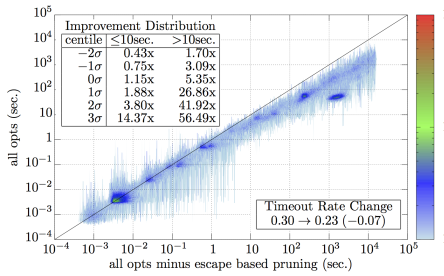
Sources and sinks. *STAR* is configured with 53 source APIs (*e.g.*, accessing user's contacts or messages) and 37 sink APIs (*e.g.*, write to network). Note that each source API may be invoked multiple times in an app. Fig. 8 shows the distribution of the number of source instructions (instructions invoking source APIs) per app. The number of source instructions easily runs into the hundreds and sometimes into the thousands. Since *STAR* is a forward analysis that begins taint tracking at the source API invocations, the number of such invocations is a key determinant of the performance.

Choice of oracles. *STAR* uses VTA [39] for call-graph construction, and a field-sensitive context-sensitive extension of Steensgaard's algorithm [38] for points-to analysis. Using a different algorithm (*e.g.*, Andersen's algorithm [2]) could affect the absolute analysis time, and it is an open question how that will change the effectiveness of the optimization techniques.

Library modeling and tainted objects. We use hand-written models for frequently used library methods. All other library methods are analyzed with a simple heuristic — the return value is considered tainted if any input parameter is tainted. If there is no return value, the receiver object is considered tainted. The implemented analysis handles tainted objects by considering any access to a field from a tainted object, as well as the return value of any method call on a tainted receiver object, as tainted. This modeling heuristic is not sound and can be improved on in the future.

[9] A reader may wonder why state-pruning techniques are necessary when SSA is available. Please note that variable level SSA is not enough to enable sparse-analysis for heap manipulating programs.

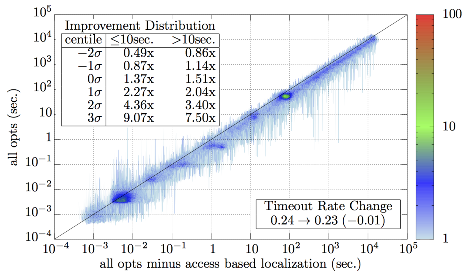
Entry point. *STAR* uses a synthesized main method that invokes a pre-defined list of entry point methods non-deterministically. The list includes lifecycle methods [10] (*e.g.*, *Activity.onCreate*), callback methods receiving intents [11] (*e.g.*, *Activity.onActivityResult*), GUI event callbacks [12] (*e.g.*, *View.onClick*), and static class initializers. This is not sound since it does not capture dependencies between arguments to these handlers. Due to the heavy use of reflection in Android and the wide variety of APIs, enumerating all entry points soundly is an open research problem.



Improvement Distribution

centile	≤10sec.	>10sec.
-2σ	0.43x	1.70x
-1σ	0.75x	3.09x
0σ	1.15x	5.35x
1σ	1.88x	26.86x
2σ	3.80x	41.92x
3σ	14.37x	56.49x

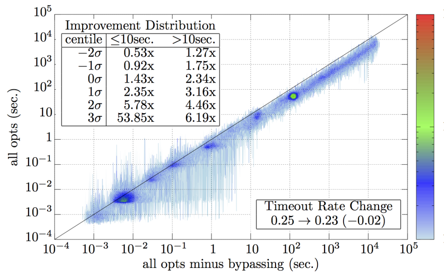
(a) Escape-based pruning.



Improvement Distribution

centile	≤10sec.	>10sec.
-2σ	0.49x	0.86x
-1σ	0.87x	1.14x
0σ	1.37x	1.51x
1σ	2.27x	2.04x
2σ	4.36x	3.40x
3σ	9.07x	7.50x

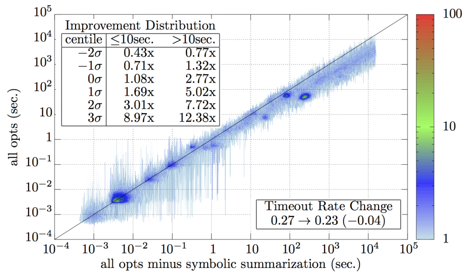
(b) Access-based localization.



Improvement Distribution

centile	≤10sec.	>10sec.
-2σ	0.53x	1.27x
-1σ	0.92x	1.75x
0σ	1.43x	2.34x
1σ	2.35x	3.16x
2σ	5.78x	4.46x
3σ	53.85x	6.19x

(c) Bypassing.



Improvement Distribution

centile	≤10sec.	>10sec.
-2σ	0.43x	0.77x
-1σ	0.71x	1.32x
0σ	1.08x	2.77x
1σ	1.69x	5.02x
2σ	3.01x	7.72x
3σ	8.97x	12.38x

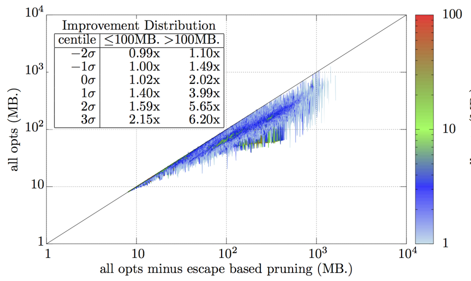
(d) Symbolic summarization.

Figure 9: Effect of optimizations on taint analysis running time. The budget was 4 hours and 10GB of memory per app. Each point represents a group of apps. The color represents the size of the group. The x-axis (resp. y-axis) measures the runtime when the corresponding optimization is disabled (resp. all optimizations are enabled).

7.3. Experiments

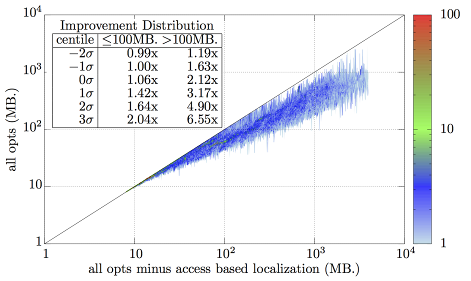
To understand the effectiveness of various optimizations (including symbolic summarization) we measured the execution time and the memory consumption of the analysis over the benchmark. We analyzed each app with a 4 hour time and 10GB memory budget. The experiment was performed using our internal computation cloud. We omit details because the company does not allow disclosure of the specifications of the internal infrastructure. We measured the

[10]https://developer.android.com/guide/components/activities/activity-lifecycle.html
[11]https://developer.android.com/guide/components/intents-filters.html
[12]https://developer.android.com/guide/topics/ui/ui-events.html



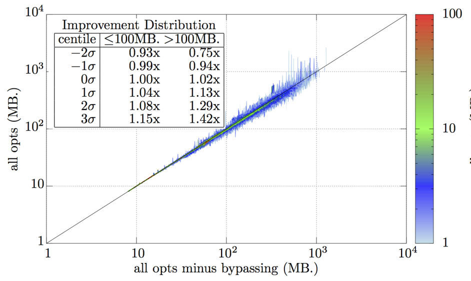
Improvement Distribution		
centile	≤100MB.	>100MB.
-2σ	0.99x	1.10x
-1σ	1.00x	1.49x
0σ	1.02x	2.02x
1σ	1.40x	3.99x
2σ	1.59x	5.65x
3σ	2.15x	6.20x

(a) Escape-based pruning.



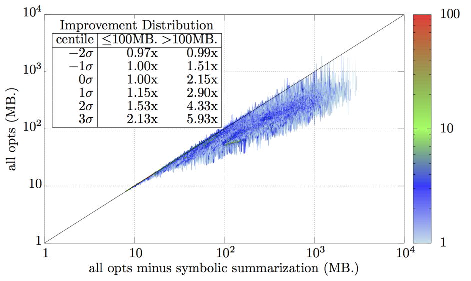
Improvement Distribution		
centile	≤100MB.	>100MB.
-2σ	0.99x	1.19x
-1σ	1.00x	1.63x
0σ	1.06x	2.12x
1σ	1.42x	3.17x
2σ	1.64x	4.90x
3σ	2.04x	6.55x

(b) Access-based localization.



Improvement Distribution		
centile	≤100MB.	>100MB.
-2σ	0.93x	0.75x
-1σ	0.99x	0.94x
0σ	1.00x	1.02x
1σ	1.04x	1.13x
2σ	1.08x	1.29x
3σ	1.15x	1.42x

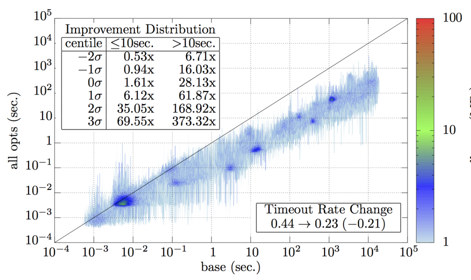
(c) Bypassing.



Improvement Distribution		
centile	≤100MB.	>100MB.
-2σ	0.97x	0.99x
-1σ	1.00x	1.51x
0σ	1.00x	2.15x
1σ	1.15x	2.90x
2σ	1.53x	4.33x
3σ	2.13x	5.93x

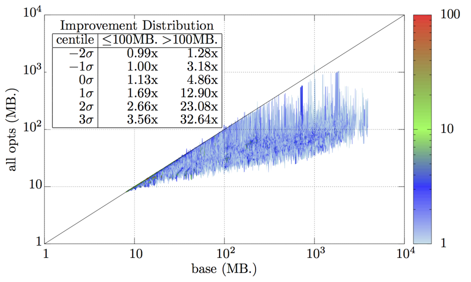
(d) Symbolic summarization.

Figure 10: Effect of optimizations on analysis peak memory consumption. The budget was 4 hours and 10GB of memory per app. Each point represents a group of apps. The color represents the size of the group. The x-axis (resp. y-axis) measures the peak memory consumption when the corresponding optimization is disabled (resp. all optimizations are enabled).



Improvement Distribution		
centile	≤10sec.	>10sec.
-2σ	0.53x	6.71x
-1σ	0.94x	16.03x
0σ	1.61x	28.13x
1σ	6.12x	61.87x
2σ	35.05x	168.92x
3σ	69.55x	373.32x

(a) Runtime.



Improvement Distribution		
centile	≤100MB.	>100MB.
-2σ	0.99x	1.28x
-1σ	1.00x	3.18x
0σ	1.13x	4.86x
1σ	1.69x	12.90x
2σ	2.66x	23.08x
3σ	3.56x	32.64x

(b) Memory consumption.

Figure 11: Effect of the full set of optimizations on analysis running time and memory consumption.

effectiveness of each optimization by disabling it, whilst enabling all other optimizations. Fig. 9 (resp. Fig. 10) show the runtime (resp. memory consumption) results. We also measured the effectiveness of *STAR* by comparing it against the naïve IFDS-version of the analysis (c.f., Fig. 11). To increase the precision of reported results, we report only the execution time of the main taint analyzer. We found that the execution time of the pre-analysis passes is negligible and on average is less than 2% of the taint analysis execution time. We also observed that the choice of optimization techniques does not have a notable impact on the running time of the pre-analyses.

7.3.1. Running Time

Fig. 9 and Fig. 11(a) show the analysis runtime results. Each subfigure is composed of three parts: (a) a scatter-plot of running time, (b) a table summarizing the distribution of the improvement (top-left), and (c) a box showing the difference in the timeout rate (bottom-right). Each point in the scatter plots represents a group of apps, where the x-coordinate (resp. y-coordinate) is the running time of the analysis without the optimization being evaluated (resp. the fully optimized analysis). Both axes are log-scaled, and the color represents the size of the group. In Fig. 11(a), the x-coordinate is the running time of the vanilla IFDS-based analysis. In each improvement distribution table, the improvement is defined as x/y (higher is better). Each table contains two data columns. The left column shows the distribution for relatively simple apps (the running time is $\leq$ 10 sec.), and the right column shows the distribution for relatively complicated apps (the running time is $>$ 10 sec.). The runtimes of simpler ($\leq$ 10 sec.) apps are noisier and less interesting, so we believe that the right column is more indicative of the value of evaluated analyses in practice. Finally, the box in the lower right corner shows the timeout rate without the optimization (left to the arrow), with the optimization (right to the arrows), and the amount of the change (in the parenthesis). The lower timeout rate is the better. Note that the timeout rates capture the information not presented in scatter plots and tables — plots and tables only show the information on apps analyzed within the time budget. Also note that a reduction in the timeout rate by 1% (i.e, 0.01) corresponds to $4,000$ more apps analyzable within the budget.

We now discuss the effects of optimizations:

- Escape-based pruning (Fig. 9(a)) is the most effective optimization technique in the combination, and the speedup tends to grow as the complexity increases. Also, the technique provides 7% reduction in the timeout rate. Such significant improvement is possible because of string values. Strings play a major role in Android taint tracking, and the way Java handles strings introduces large number of temporary objects. For instance, each string concatenation operation using the + operator introduces two new objects: a temporary *StringBuilder* type object and the resulting string object. If a string is gradually constructed across multiple expressions, such a construction will introduce a large number of method-local objects (all temporary *StringBuilder* objects and intermediate strings constructed during the construction process). We found that a significant portion of the taint facts generated by the baseline taint analysis are about such temporary *StringBuilder*s and strings, and the escape-based pruning can effectively remove them.
- Symbolic summarization (Fig. 9(d)) also offers significant performance improvements. The effectiveness of symbolic summarization can vary depending on the number of source instructions in a target program. A larger

app is more likely to use more such instructions. Thus, the technique will be increasingly effective when target apps become more complex, as indicated by results and 4% reduction in the timeout rate. We also analyzed a set of 30 apps and verified that the speedup provided by the symbolic summarization technique was positively correlated to the number of active sources. Overall, the results show that symbolic summarization technique provides significant speedups, comparable to escape-based pruning. Most importantly, the speedup is additive, *i.e.*, symbolic summarization can be synergistically combined with other optimization techniques.

- Access-based localization (Fig. 9(b)) and bypassing (Fig. 9(c)) show similar pattern on the complicated apps — the ratio of improvement seems to be limited by a constant factor. Also, neither of those two techniques is very effective at significantly reducing timeouts (1% and 2% each). Overall, bypassing seems to be somewhat more effective of the two.
- The fully-optimized analysis runs orders of magnitude faster than the vanilla implementation (Fig. 11(a)). The distribution shows that, for complicated applications, more than 81.8% of apps (between -1σ and 2σ) get an order of magnitude speedup, and the right tail (above 2σ) enjoys two orders of magnitude of speedup. Some extreme apps even get 3-4 orders of magnitude of speedup. The reduction of the timeout rate is also significant (21%).

7.3.2. Memory Consumption

Fig. 10 and Fig. 11(b) show the analysis memory consumption results. The plots are similar to the runtime plots, except that x-coordinates and y-coordinates indicate the peak memory consumption, instead of the analysis runtime. In tables, the left data column shows the distribution for relatively small-footprint apps (the memory consumption is $\leq$ 100 MB), and the right column shows the distribution for relatively large-footprint apps (the memory consumption is $>$ 100 MB.). We now discuss the effects of optimizations:

- Access-based localization (Fig. 10(b)), escape-based pruning (Fig. 10(a)), and symbolic summarization (Fig. 10(d)) all reduce the memory consumption. All three are showing similar improvement distributions and similar patterns — they become more effective as the peak memory consumption increases.
- Bypassing (Fig. 10(c)) has a mixed impact on the memory consumption. It increases the memory consumption of half of the apps, and it reduces the memory consumption of the other half. When the memory consumption is an important constraint, the bypassing technique should be used in combination with other optimizations in order to cancel out the negative effect of the technique. There are two reasons why bypassing does not decrease the memory consumption significantly, and can even increase it occasionally.
    - Bypassing itself does not save a lot of memory as it only decreases the size of taint fact sets reaching non-invoke instructions. The size

of taint fact sets reaching invoke-instructions and the size of method-summaries remain unchanged, although the amount of computation required to compute them is decreased. Overall, the memory saving is proportional to the number of non-invoke instructions and the number of non-local taint-facts reaching each method.

  – Bypassing has an additional bookkeeping overhead as it needs to know the immediate post-dominating invoke instruction for each instruction. The cost is proportional to the number of instructions (including invokes) in each method.

  The combination of these two factors determines whether the memory consumption will increase or decrease. The results suggest that the benefit and the cost cancel out each other most of the time, and the overall memory consumption does not change significantly. However, if an app has several methods containing a large number of invoke-instructions, the cost can exceed the benefit.

- The fully-optimized analysis (Fig. 11(b)) reduces memory consumption for most applications. For half of the apps (below the median), it provides a moderate improvements (1.28x–4.86x). Above the median, it provides a significant improvement (4.86x–32.64x).
- One interesting observation is that the memory consumption never exceeds the 10GB budget. This is due to the fact that *STAR* implementation aggressively uses hash-consing [31, 1] to reduce memory consumption.

8. Related Work

8.1. Flow-Sensitive Summary Based Analysis

IFDS. IFDS [34] and IDE [36] frameworks are well-known instances of applying summarization to flow-and-context-sensitive program analysis. IFDS has been applied to type-state analysis [12], error diagnosis [24], and taint analysis [41, 37, 3]. In Section 2, we discussed the differences between *STAR* and IFDS/IDE in detail.

FlowTwist [17, 18] is an IFDS based Android/Java taint analyzer. FlowTwist implements two important optimization techniques; one is comparable to our symbolic summarization and another is orthogonal.

- FlowTwist uses symbolic reasoning [17] to resolve the redundancies caused by tracking multiple taint sources. The main difference is in the way FlowTwist determines actual taint sources that reach sinks: FlowTwist tracks causal relationship between taint facts while propagating them and uses this information to backtrack the taint sources reaching each sink instruction at the end. In contrast, our technique computes summaries with effects, which help the analysis to determine reaching taint sources without performing back-tracking. The information computed by FlowTwist is richer in the sense that it can be used to backtrack arbitrary taint facts at arbitrary program points, at the cost of more bookkeeping.

- FlowTwist uses access-path-abstraction [18] that allows grouping a set of access-paths and using a single taint fact for the entire group. This is useful to handle identity transfer functions. Without access-path-abstraction, the transfer function for an instruction needs to have the identity propagation entry for each access-path that is not affected by the instruction. With access-path-abstraction, the transfer function for such an instruction can be represented succinctly by having one identity propagation per group. Note that this is orthogonal to the symbolic summarization used in *STAR*: our technique symbolically handles sets of taint sources, whereas access-path-abstraction symbolically handles sets of access-paths. Hence, one should be able to combine both techniques.

Naeem and Lhotak [26] proposed extensions to IFDS mainly focused on improving efficiency. One novel improvement was to generate inter-procedural supergraph edges on demand. Rapoport *et al.* [33] proposed a technique to suppress infeasible paths in the context of object-oriented programs by recognizing correlated methods calls. This work considers two method calls as correlated if they are known to have the same receiver object. It relies on identifying such calls to improve precision by ignoring infeasible information flow. Both ideas [26, 33] can be combined with *STAR*.

Memory analysis. Summarization has been applied to flow-and-context sensitive memory-analysis problems [8, 32, 43, 9, 13]. Those analyses do not satisfy the requirements of the IFDS framework, and each of them proposes their own summarization methods, whereas summaries used by *STAR* are an extension of IFDS. Padhye and Khedker [32] develop a tabulation approach suited for non-distributive analysis, and apply to points-to analysis.

CFA2 [43] and pushdown-CFA [9, 13] apply a similar idea to control flow analysis. Those techniques do not decompose summaries, which could limit their scalability in practice. Dillig *et al.* [8] propose a summarization approach to alias-analysis with support for strong updates. The symbolic summaries in their work are based on a sophisticated decomposition suited for alias analysis.

Introspective pushdown-CFA [9] combines pushdown-CFA [9, 13] and abstract garbage collection [25]. The key idea is to track the root-sets of reachable abstract locations as a part of summaries. It is unclear whether the idea is applicable to *STAR* without breaking distributivity of the transfer functions.

Value analysis. In the context of flow and context-sensitive value analysis, two important studies [16, 45] have been reported recently. Lazy-propagation [16] is a top-down analysis algorithm computing generalized-summaries, and it can be applied to non-IFDS problems. When analyzing a function, the lazy-propagation algorithm first analyzes the function with a purely symbolic abstract state, figures out what details of the symbolic abstract value are necessary during the analysis, refines the input symbolic abstract value according to the observation, and analyzes the function again. This allows the algorithm to find the right amount of details required to analyze each function and to create generalized summaries based on the findings. Zhang *et al.* [45] also proposed an analysis algorithm that computes summaries containing the right amount of details. The proposed algorithm uses a top-down analysis as the baseline, and switches to the symbolic bottom-up analysis to compute a generalized-summary for a certain function only if the function is analyzed several times. When computing a generalized summary, the symbolic analysis takes one of the observed abstract states as a witness input and computes the partial symbolic summary capturing the part of the function behavior relevant to the given witness. It is an open research question whether *STAR* can benefit from these techniques. Note that these techniques are adaptive methods that are useful when an optimal summary generalization strategy cannot be determined statically. On the contrary, it is always better to use the most general summary in *STAR*.

8.2. Android Security Analysis

There has been a considerable amount of work on dynamic (*e.g.*, [10]) and static taint analysis to identify security issues in Android apps. We focus on the latter.

One thread of work on static taint analysis has focused on on-demand alias analysis interleaved with taint analysis. Andromeda [42] presents such a technique for scaling taint analysis for Java applications avoiding an expensive points-to pre-analysis. FlowDroid [3] also performs interleaved points-to analysis with the notion of activation contexts, which gives a precise way to manage the handover between the two analyses. *STAR* relies on pre-computed points-to analysis since it is aimed at analyzing apps with a high density of taint sources.

A second area of work has been to model the semantics of Android framework (particularly the control flow) with greater precision. DroidSafe [14] proposes information-flow analysis that models the Android APIs with carefully written stubs (for component interaction as well as native methods), which allows the analysis to handle callbacks. Apposcopy [11] implements high-level malware detection rules over an inter-component call-graph. IccTA [20] identifies privacy leaks across multiple Android components by relying on a suite of analyses to resolve inter-component communication. The techniques in *STAR* are orthogonal to those in DroidSafe, Apposcopy, and IccTA since our focus is on improving performance of the fixed point computation. Our techniques can be applied over the control flow graph inferred by those techniques.

Finally, Liang *et al.* [21] recently proposed a static taint analyzer for Android apps based on Pushdown CFA and entry-point saturation. That work presents the details of the formalism, but does not evaluate it experimentally. Thus, we are unable to compare it to *STAR*.

Vitae

Wontae Choi. Wontae Choi is a software engineer at Google, Inc. He currently focuses on analyzing Android applications using static and dynamic analysis. Previously, Wontae worked on automated test generation, type system, and static program analysis. He received B.S. and M.S. in Computer Science from

Seoul National University in 2008 and 2010. He received his Ph.D in Computer Science from University of California, Berkeley in 2017.

Jayanthkumar Kannan. Jayanthkumar Kannan obtained his PhD from UC Berkeley, and then worked at Google Security.

Domagoj Babic. Domagoj Babic is a manager and a tech lead at Google, Inc. His work focuses on research and development of automated software analysis systems. Over his career, Domagoj has published in the areas of verification, testing, security of complex software systems, automated reasoning, grammar inference, and applied formal methods. Before joining Google, Domagoj was a research scientist at UC Berkeley and elsewhere in industry. He received his Dipl.Ing. in Electrical Engineering and M.Sc. in Computer Science from the Zagreb University in 2001 and 2003. He received his Ph.D. in Computer Science in 2008 from the University of British Columbia.

[1] J. Allen. *Anatomy of LISP*. McGraw-Hill, Inc., 1978.

[2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[4] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proc. of the 13th International Conference on Static Analysis*, SAS'06, pages 221–239. Springer-Verlag, 2006.

[5] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 97–103. ACM, 2001.

[6] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proc. of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 20–34, 1999.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, 1999.

[8] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proc. of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 567–577, 2011.

[9] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188, 2012.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[11] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, 2014.

[12] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[13] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. Van Horn. Pushdown control-flow analysis for free. In *Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, 2016.

[14] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *Proc. of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2014*, 2015.

[15] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proc. of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, 2011.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, pages 320–339. Springer, 2010.

[17] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–108. ACM, 2014.

[18] J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-path abstraction: scaling field-sensitive data-flow analysis with unbounded access paths (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 619–629. IEEE, 2015.

[19] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 343–353, 2011.

[20] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proc. of the 37th International Conference on Software Engineering — Volume 1*, ICSE '15, pages 280–291. IEEE Press, 2015.

[21] S. Liang, M. Might, and D. Van Horn. AnaDroid: Malware analysis of Android with user-supplied predicates. *Electronic Notes in Theoretical Computer Science*, 311:3–14, 2015.

[22] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, 1988.

[23] M. Madsen and A. Møller. Sparse dataflow analysis with pointers and reachability. In *Proc. of the 21st International Static Analysis Symposium*, SAS '14, pages 201–218. Springer-Verlag, 2014.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *Proc. of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 63–72, 2004.

[25] M. Might and O. Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proc. of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 13–25, 2006.

[26] N. A. Naeem, O. Lhoták, and J. Rodriguez. Practical extensions to the IFDS algorithm. In *Proc. of the 19th International Conference on Compiler Construction*, CC '10, pages 124–144. Springer, 2010.

[27] H. Oh and K. Yi. Access-based localization with bypassing. In *Proc. of the 9th Asian Symposium on Programming Languages and Systems*, APLAS '11, pages 50–65. Springer, 2011.

[28] H. Oh, L. Brutschy, and K. Yi. Access analysis-based tight localization of abstract memories. In *Proc. of the 12th International Conference on Verification, Model Checking, and Abstract*, VMCAI '11, pages 356–370. Springer, 2011.

[29] H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi. Global sparse analysis framework. *ACM Trans. Program. Lang. Syst.*, 36(3):8:1–8:44, Sept. 2014.

[30] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19. Springer, 2001.

[31] C. Okasaki and A. Gill. Fast mergeable integer maps. In *In ACM Workshop on ML*, pages 77–86, 1998.

[32] R. Padhye and U. P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proc. of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, 2013.

[33] M. Rapoport, O. Lhoták, and F. Tip. Precise data flow analysis in the presence of correlated method calls. In *Proc. of the 22nd International Symposium on Static Analysis*, SAS '15, pages 54–71. Springer, 2015.

[34] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, 1995.

[35] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 285–293, 1988.

[36] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, pages 131–170. Elsevier Science Publishers B. V., 1996.

[37] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. pages 1053–1068, 2011.

[38] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, 1996.

[39] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. *Practical virtual method call resolution for Java*, volume 35. ACM, 2000.

[40] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[41] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, 2009.

[42] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proc. of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 210–225. Springer-Verlag, 2013.

[43] D. Vardoulakis and O. Shivers. CFA2: A context-free approach to control-flow analysis. In *Proc. of the 19th European Symposium on Programming Languages and Systems*, ESOP '10, pages 570–589. Springer, 2010.

[44] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, 1999.

[45] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 249–258, 2014.