Automated Inference of Library Specifications for Source-Sink Property Verification*

Haiyan Zhu[1], Thomas Dillig[2], and Isil Dillig[3]

[1] College of William & Mary
[2] University College London
[3] Microsoft Research Cambridge

Abstract. Many safety properties in program analysis, such as many memory safety and information flow problems, can be formulated as *source-sink* problems. While there are many existing techniques for checking source-sink properties, the soundness of these techniques relies on all relevant source code being available for analysis. Unfortunately, many programs make use of libraries whose source code is either not available or not amenable to precise static analysis. This paper addresses this limitation of source-sink verifiers through a technique for inferring exactly those library specifications that are needed for verifying the client program. We have applied the proposed technique for tracking explicit information flow in Android applications, and we show that our method effectively identifies the needed specifications of the Android SDK.

1 Introduction

Many safety properties of interest in program analysis can be formulated in terms of verifying the absence of *source-sink* errors. Such an error arises if a value constructed at a location designated as a *source* reaches a location designated as a *sink*. Examples of source-sink problems include the following:

- Confidential information (source) cannot be sent to an untrusted party (sink).
- A pointer assigned to null (source) should not reach a dereference (sink).
- A closed file f (source) should not be read or written (sink).

Over the last decade, there has been much progress in verifying the absence of source-sink errors [1–3]. Given a value v constructed at source location l_1, and a value w consumed at sink location l_2, source-sink checkers determine if there exists a feasible execution path from l_1 to l_2 on which v and w are equal. As an example, consider the following Java-like code:

```
1. Data d = new Data();
2. Location x = null;
3. if(R) x = getGPSLocation();
4. d.loc = x ;
5. if(R) send(d.loc, "http://xue.com/stealmyloc.php");
```

Here, we want to determine whether confidential data can be sent over the network. The method `getGPSLocation` is a source since it returns the user's

* This work is supported in part by DARPA #FA8750-12-2-0020

confidential GPS location. In contrast, the method `send(x,y)` called at line 5 is a sink because it sends data `x` to URL `y`. Assuming predicate `R` can be true, the above code snippet has a source-sink error because there is a feasible execution path from the source to the sink in which `x` and `d.loc` are aliases.

While automated source-sink checkers have improved substantially in terms of precision and scalability over the last decade, they typically make two assumptions to guarantee soundness: First, they require sources and sinks to be specified by the user. Second, they require all relevant source code to be available for analysis. The first requirement is often not too cumbersome because there are typically few kinds of sources and sinks, and there has been recent progress on automating source and sink inference [4]. On the other hand, the second assumption is more problematic because modern software uses of many layers of complex libraries. While calls to library methods can –and often do– affect source-to-sink flows, it is often impractical to analyze library code together with the client, for example, because library code may be unavailable or may be written in a different language. Furthermore, even when library code is available, its implementation is typically much larger and much lower-level than the client, making it undesirable to analyze the library's implementation for verifying the client. Existing source-sink checkers deal with this difficulty in one of three ways:

1. Assume an *angelic* environment by treating library calls as no-ops. Unfortunately, this amounts to the optimistic but unsound assumption that library methods do not introduce flows from sources to sinks.
2. Assume a *demonic* environment by making worst-case assumptions about library methods, which means that a library method m may introduce a flow between any pair of locations reachable in m.
3. Require the user to write *flow specifications* of library methods, which describe whether input x may transitively reach output y.

Unfortunately, all of these options have serious drawbacks. The first option is unsound and yields many false negatives. The second option is sound but grossly imprecise, yielding many false positives. Finally, the third alternative is extremely cumbersome for users: In modern software, there are typically *many* calls to library functions, each of which could require several flow annotations. In principle, not all of these flow specifications are relevant for verifying the absence of source-sink errors, but it is very difficult for humans to reliably identify which specifications are needed to guarantee soundness of the analysis.

In this paper, we address this limitation of source-sink checkers by automating the inference of library specifications that are needed for soundness. Given a client program A, our technique infers a *smallest* set of *must-not-flow* requirements on library functions that are sufficient to ensure that A is free of source-sink errors. Since our technique only analyzes implementations of clients but not libraries, specifications inferred by our technique must be checked against either the documentation or the implementation of the library. However, as we show experimentally, the number of must-not-flow requirements inferred by our technique are only a small fraction of the possible flow relations that are possible; hence, our technique minimizes the effort required to guarantee soundness.

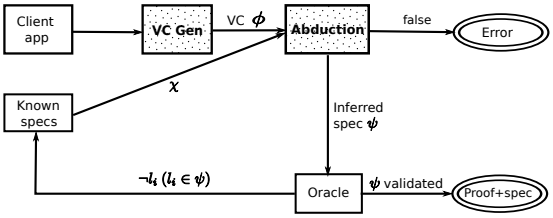

Fig. 1. A schematic illustration of our approach

1.1 Overview

The high-level architecture of our approach is shown in Figure 1. Given a client application A, our technique first analyzes A to generate a verification condition (VC) ϕ. This VC is parametrized over the possible flows that can be introduced due to library calls and its validity guarantees the absence of source-sink errors in A. Given such a VC ϕ and a formula χ encoding known partial specifications of library methods, our technique enters a refinement loop with the "Abduction" component at its core. The formula χ is initially just true, meaning there are no known specifications of library methods, but becomes logically stronger as the refinement process continues.

At every step of the refinement loop, we use an inference technique known as *abduction* to speculate a candidate specification ψ, which asserts a minimal set of must-not-flow requirements on library methods that are sufficient to guarantee program A's correctness. Since our technique does not analyze library implementations, each must-not-flow requirement in ψ must be externally validated by an oracle. This oracle can be a user who can consult the documentation of the library or a different technique for analyzing the library's source code or binary. In either case, since we want to minimize the amount of work to be performed by the oracle, our inferred candidate specification ψ should be as small as possible.

Given the candidate specification ψ, we then ask the oracle to confirm or refute each must-not-flow requirement l_i in ψ. If the oracle can confirm each $l_i \in \psi$, we have found a correct and minimal specification sufficient to verify A. In this case, the refinement loop terminates with $\psi \Rightarrow \phi$ as a proof of correctness of A. On the other hand, if the oracle cannot certify some must-not-flow relation l_i from α to β in ψ, this means there is a may-flow relation from α to β. Therefore, the negation of l_i is added to the set of known specifications χ, and abduction is used again to infer a different candidate specification ψ'. This process continues until either we find a correct specification sufficient to verify A or until we prove that there is no correct must-not-flow specification of the library sufficient to discharge the VC ϕ. We now give a brief overview of the two key components underlying our technique, illustrated as "VC Gen" and "Abduction" in Figure 1.

VC Generation Our approach to generating VCs is based on the following insight: Rather than making purely angelic or purely demonic assumptions, we introduce *constraints* describing the possible effects of library methods. These constraints are composed of boolean *flow variables* $f_{l_1 \mapsto l_2}$, which describe whether the value in location l_1 *may* flow to location l_2 in library function f. As an example, consider the statement x = f(y) where f is a library method. Here, if y was tainted before the call, our analysis will taint x under the constraint $f_{a_1 \mapsto \text{ret}}$,

where the boolean variable $f_{a_1 \mapsto \text{ret}}$ represents whether f's first argument may (transitively) flow to its return value.

Now, when we encounter a sink, we generate a VC that is parametric over these flow variables. Specifically, if a variable x used at a sink has value v under constraint φ, then the generated VC asserts that φ implies that v is not tainted. Since the value constraints φ are parametrized over possible flow relations, the validity of the VC therefore depends on the truth assignment to flow variables.

The advantage of this strategy is that the generated VC captures the full range of possible assumptions on the library in between the angelic and demonic ones. On one extreme, if the VC is valid, the client application can be verified even under demonic assumptions. On the other hand, if the VC is unsatisfiable, the client application cannot be verified even with angelic assumptions about library calls. However, if the formula is contingent, the VC still contains useful information about flow specifications needed to discharge the source-sink flow.

Abductive Inference The second insight underlying our technique is that *logical abduction* can be used to infer a minimal set of must-not-flow requirements that are needed to guarantee the absence of source-sink errors in the client application. Specifically, given two formulas ϕ and χ, logical abduction is the problem of finding an explanatory hypothesis ψ such that:

$$(1)\ \ \chi \wedge \psi \models \phi \quad \text{and} \qquad (2)\ \text{SAT}(\chi \wedge \psi)$$

In our setting, ϕ corresponds to the VC generated for the client, and χ corresponds to the known assumptions on library methods. Therefore, the first condition says that, together with known specifications χ, the solution ψ to the inference problem should imply the verification condition ϕ. The second condition says that the abductive solution ψ should not contradict known specifications χ.

In addition to these two requirements, we want our solution ψ to be the smallest conjunction of flow literals satisfying the above conditions. This requirement is important since we want to minimize the number of assumptions to be externally validated by an oracle. For this purpose, our technique uses *minimum-size prime implicants* of boolean formulas to compute the desired abductive solutions.

1.2 Organization and Contributions

The rest of this paper is organized as follows: Section 2 gives the syntax and semantics of a small language used for our formalization. Section 3 describes a sound static analysis for generating VCs parametric over flow variables, and Section 4 describes our inference algorithm based on minimum-size prime implicants. Sections 5 and 6 describe important extensions and our implementation. Finally, Sections 7, 8, 9 describe experimental results, related work, and future directions. To summarize, this paper makes the following key contributions:

- We present a novel technique for inferring flow specifications of libraries for source-sink property verification.
- We show how to generate verification conditions that are parametric over the unknown behavior of library methods.

$$
\begin{array}{ll}
\text{Program } P & := m\ f^* \\
\text{Client } m & := \text{def } m = \{\kappa\} \\
\text{Library function } f & := \text{def } f(\alpha_1, \ldots, \alpha_n) = \{\varsigma;\ \chi_f \leftarrow v\} \\
\text{Client stmt } \kappa & := v \leftarrow \mathcal{S} \mid \text{check}(v) \mid a \mid \kappa_1; \kappa_2 \mid \text{if}(\star) \text{ then } \kappa_1 \text{ else } \kappa_2 \mid v \leftarrow^{\rho} f(v_1, \ldots, v_n) \\
\text{Library stmt } \varsigma & := a \mid \varsigma_1; \varsigma_2 \mid \text{if}(\star) \text{ then } \varsigma_1 \text{ else } \varsigma_2 \\
\text{Assignment } a & := v \leftarrow c \mid v_1 \leftarrow v_2 \\
\text{Constant } c & := \mathcal{C}_1 \mid \ \ldots \ \mid \mathcal{C}_k
\end{array}
$$

Fig. 2. Language used for formal development

- We formulate the minimum flow specification inference problem as an instance of logical abduction and give an algorithm based on minimum-size prime implicants for solving the generated inference problems.
- We apply the proposed technique for verifying confidentiality of Android applications that heavily use unanalyzed library methods. Experimentally, we show that our method is effective at identifying a small set of relevant flow specifications that are needed for analyzing the client.

2 Language and Concrete Semantics

Figure 2 defines an imperative call-by-value language used for our formalization. In this language, a program consists of one client application m and zero or more library functions f. The client m has body κ, and our goal is to verify m without analyzing libraries called by m. While our technique will not analyze library functions f, we give their syntax and semantics in order to precisely define what we mean by their flow specifications.

In this language, the special constant $\mathcal{S}$ denotes a *source*; hence, the assignment $v \leftarrow \mathcal{S}$ *taints* variable v. The statement check(v) is a *sink*: It evaluates to false if v is tainted (i.e., value of v is $\mathcal{S}$); otherwise, it evaluates to true. If the check statement check(v) evaluates to false, we say that the check *fails*.

In client m, statements are sources, sinks, assignments, sequencing, conditionals, and calls to library functions, which are annotated with a unique *label* ρ. Library functions f can take any number of arguments $\alpha_1, \ldots, \alpha_n$ and consist of a body ς and a statement $\chi_f \leftarrow v$, where χ_f denotes the return value of f. Statements ς used in library functions are the same as those used in the client except that they cannot be sources or sinks, since we assume library methods corresponding to sources and sinks are annotated. Hence, while library functions can propagate taint, they neither generate nor leak tainted values.

To focus on the novel ideas underlying our technique, our formal development intentionally omits pointers. Section 5 will explain how the proposed technique can reason about flows between objects in the heap.

2.1 Operational Semantics

To allow providing a soundness proof of our approach, Figure 3 presents a large-step operational semantics of the language from Figure 2. The operational semantics are described using judgments of the form $\Pi, \Gamma \vdash s : \Gamma', b$. Here, Π maps each function name to its definition, and the store Γ maps each variable to its value at run-time. A new store Γ' is obtained by executing statement s starting with store Γ, and the boolean value b indicates whether there is a failing check statement in s. Therefore, rules (4a) and (4b) for check(v) statements

$$(1)\frac{\Gamma' = \Gamma[v \mapsto c]}{\Pi, \Gamma \vdash v \leftarrow c : \Gamma', \text{true}} \quad (2)\ \frac{\Gamma(v_2) = c \quad \Gamma' = \Gamma[v_1 \mapsto c]}{\Pi, \Gamma \vdash v_1 \leftarrow v_2 : \Gamma', \text{true}} \quad (3)\ \frac{\Pi, \Gamma \vdash\ s_1 : \Gamma_1, b_1 \quad \Pi, \Gamma_1 \vdash\ s_2 : \Gamma_2, b_2}{\Pi, \Gamma \vdash s_1; s_2 : \Gamma_2, b_1 \wedge b_2}$$

$$(4a)\ \frac{\Gamma(v) = c \quad c \neq \mathcal{S}}{\Pi, \Gamma \vdash \text{check}(v) : \Gamma, \text{true}} \quad (4b)\ \frac{\Gamma(v) = c \quad c = \mathcal{S}}{\Pi, \Gamma \vdash \text{check}(v) : \Gamma, \text{false}}$$

$$(5a)\ \frac{(s_1 \oplus s_2) = s_1 \quad \Pi, \Gamma \vdash s_1 : \Gamma_1, b_1}{\Pi, \Gamma \vdash \text{if}(\star) \text{ then } s_1 \text{ else } s_2 : \Gamma_1, b_1} \quad (5b)\ \frac{(s_1 \oplus s_2) = s_2 \quad \Pi, \Gamma \vdash s_2 : \Gamma_2}{\Pi, \Gamma \vdash \text{if}(\star) \text{ then } s_1 \text{ else } s_2 : \Gamma_2, b_2}$$

$$(6)\ \frac{\begin{array}{c}\Pi(f) = \lambda\alpha_1, \ldots, \alpha_n.\{s; \chi_f \leftarrow v\} \\ \Gamma(v_1) = c_1 \ \ldots \Gamma(v_n) = c_n \\ \Pi, [\alpha_1 \mapsto c_1, \ldots, \alpha_n \mapsto c_n] \vdash \{s; \chi_f \leftarrow v\} : \Gamma' \\ \Gamma'(\chi_f) = c\end{array}}{\Pi, \Gamma \vdash v \leftarrow^{\rho} f(v_1, \ldots, v_n) : \Gamma[v \mapsto c, \pi_\rho \mapsto c], \text{true}}$$

Fig. 3. Operational semantics

produce true or false depending on whether v is $\mathcal{S}$. In rules (5a) and (5b) for if statements, the notation $s_1 \oplus s_2$ non-deterministically chooses either s_1 or s_2.

Rule (6) in Figure 3 gives the semantics of calls to library methods. Since this language has call-by-value semantics, only the value of variable v can change in the client code as a result of the call $v \leftarrow^{\rho} f(v_1, \ldots, v_n)$. Specifically, since χ_f denotes function f's return value, v is assigned to c whenever χ_f evaluates to c. The variable π_ρ used in rule (6) is an instrumentation variable and is only introduced to facilitate the soundness proof of our abstract semantics.

Definition 1 *A concrete execution of a program P has a* source-sink error *if and only if P evaluates to false.*

2.2 Flows in Concrete Executions

Since our technique will infer flow specifications of library functions, we first formally define what we mean by a *flow* in a concrete execution. In this section, we represent a concrete execution of a program by a *trace* $\sigma = s_1, s_2, \ldots, s_n$ consisting of the sequential execution of *instructions* s_1 through s_n. Instructions can be assignments, check statements, function invocations, or function returns. We write $\text{call}^{\rho}\ f(v_1, \ldots, v_n)$ to denote the invocation of function f with actuals $v_1, \ldots, v_n$ at a call site $v \leftarrow^{\rho} f(v_1, \ldots, v_n)$, and we write $\text{return}^{\rho}\ f \mapsto v$ to indicate f's return and the assignment of its return value to variable v.

Given a trace $\sigma = s_1, s_2, \ldots, s_n$, we write s_i^+ and s_i^- to represent the control points *right after* and *right before* the execution of instruction s_i respectively. Observe that this implies $s_i^+ = s_{i+1}^-$. We can now define a *one-step flow* relation $\rightsquigarrow$ between variables u, v on a concrete execution σ:

Definition 2 *Let $\sigma = s_1, s_2, \ldots, s_n$ be an execution trace. We define the one-step flow relation $\rightsquigarrow$ to be the smallest relation satisfying the following conditions:*

- $(u, s_i^-) \rightsquigarrow (u, s_i^+)$ *if* $s_i \neq (u \leftarrow \ldots)$
- $(u, s_i^-) \rightsquigarrow (v, s_i^+)$ *if* $s_i = (v \leftarrow u)$

The first condition here states that $(u, s_i^-) \rightsquigarrow (u, s_i^+)$ if s_i does not reassign u. The second condition says that $(u, s_i^-) \rightsquigarrow (v, s_i^+)$ if instruction s_i is an assignment from u to v. Thus, intuitively, the one-step flow relation $(u, s_i^-) \rightsquigarrow (v, s_i^+)$ encodes whether the value of variable u flows to variable v in instruction s_i. Using the relation $\rightsquigarrow$, we now define a *multi-step flow relation* $\rightsquigarrow^*$ as follows:

Definition 3 *The multi-step flow relation $\rightsquigarrow^*$ is the smallest relation satisfying the following conditions:*

- $(u, s_i^-) \rightsquigarrow^* (v, s_j^+)$ *if* $(u, s_i^-) \rightsquigarrow (v, s_j^+)$
- $(u, s_i^-) \rightsquigarrow^* (w, s_k^+)$ *if* $(u, s_i^-) \rightsquigarrow^* (v, s_j^+)$ *and* $(v, s_{j+1}^-) \rightsquigarrow^* (w, s_k^+)$
- $(v_k, s_i^-) \rightsquigarrow^* (v, s_j^+)$ *if* $s_i = \text{call}^\rho f(v_1, \ldots v_n)$ *and* $s_j = \text{return}^\rho f \mapsto v$ *and* $(\alpha_k, s_{i+1}^-) \rightsquigarrow^* (\chi_f, s_{j-1}^+)$

If $(u, s_i^-) \rightsquigarrow^* (v, s_j^+)$, we say that u at s_i *flows to* v at s_j. The first two conditions in Definition 3 state that $\rightsquigarrow^*$ includes the transitive closure of $\rightsquigarrow$. The third condition deals with flows that are introduced due to calls to library functions. Specifically, consider a pair (s_i, s_j) of matching function call/return instructions where $s_i = \text{call}^\rho\ f(v_1, \ldots, v_n)$ and $s_j = \text{return}^\rho f \mapsto v$. Observe that the sub-trace given by $s_{i+1}, s_{i+2}, \ldots, s_{j-1}$ corresponds to the execution of f's body. According to the third rule of Definition 3, if the k'th argument of f flows to the return value of f between s_{i+1} and s_{j-1}, then the value of the actual v_k before instruction s_i also flows to variable v after instruction s_j.

Example 1. Consider the following trace σ:

$$s_1 : u \leftarrow \mathcal{S} \qquad s_2 : v \leftarrow u \qquad s_3 : \text{call}^\rho f(v) \qquad s_4 : x \leftarrow \alpha_1$$
$$s_5 : \chi_f \leftarrow x \qquad s_6 : \text{return}^\rho f \mapsto w \qquad s_7 : z \leftarrow w \qquad s_8 : \text{check}(z)$$

Here, $(\alpha_1, s_4^-) \rightsquigarrow^* (\chi_f, s_5^+)$ because f's first argument is transitively assigned to its return value. Due to the call between s_3 and s_6, $(v, s_3^-) \rightsquigarrow^* (w, s_6^+)$. Finally, since $(u, s_1^+) \rightsquigarrow^* (v, s_3^-)$ and $(w, s_6^+) \rightsquigarrow^* (z, s_8^-)$, we have $(u, s_1^+) \rightsquigarrow^* (z, s_8^-)$.

Since we will use flow specifications of library functions to determine the absence of source-sink errors, it is helpful to give the following alternate characterization of source-sink errors in terms of flows:

Proposition 1. *Trace $\sigma = s_1, s_2, \ldots, s_n$ has a source-sink error iff there exists some $s_i, s_j \in \sigma$ such that $s_i : (u \leftarrow \mathcal{S})$ and $s_j : \text{check}(v)$ and $(u, s_i^+) \rightsquigarrow^* (v, s_j^-)$.*

Example 2. The trace from Example 1 has a source-sink error because $(u, s_1^+) \rightsquigarrow^* (z, s_8^-)$ and s_1 is $u \leftarrow \mathcal{S}$ and s_8 is check(z).

3 Analysis and VC Generation

This section describe a static analysis for generating *verification conditions* that, if valid, imply the absence of source-sink errors. As mentioned earlier, the VCs generated by our analysis are parametrized over boolean *flow variables*, which soundly model the unknown effects of calls to library functions.

The VC generation procedure is described as inference rules shown in Figure 4. These rules use an environment Ω, which is the abstract counterpart of

$$(1)\ \frac{\Omega' = \Omega[v \mapsto (c, \text{true})]}{\Omega \vdash v \leftarrow c : \Omega', \text{true}} \qquad (2)\ \frac{\Omega(v_2) = \theta\ \Omega' = \Omega[v_1 \mapsto \theta]}{\Omega \vdash v_1 \leftarrow v_2 : \Omega', \text{true}}$$

$$(3)\frac{\begin{array}{c}\Omega\ \vdash\ s_1 : \Omega_1, \phi_1 \\ \Omega_1 \vdash\ s_2 : \Omega_2, \phi_2\end{array}}{\Omega \vdash s_1;^{\rho} s_2 : \Omega_2, \phi_1 \wedge \phi_2} \qquad (4)\frac{\begin{array}{c}\Omega(v) = \theta \\ \phi = \bigwedge\limits_{(\mathcal{A}_i, \phi_i) \in \theta} \phi_i \Rightarrow (\mathcal{A}_i \neq \mathcal{S})\end{array}}{\Omega \vdash \text{check}(v) : \Omega, \phi}$$

$$(5)\frac{\begin{array}{c}\Omega \vdash s_1 : \Omega_1, \phi_1 \\ \Omega \vdash s_2 : \Omega_2, \phi_2\end{array}}{\Omega \vdash \text{if}(\star) \text{ then } s_1 \text{ else } s_2 : \Omega_1 \sqcup \Omega_2, \phi_1 \wedge \phi_2}$$

$$(6)\frac{\begin{array}{c}\theta_i = \{(\mathcal{A}_{ij}, \phi_{ij} \wedge f_i) \mid (\mathcal{A}_{ij}, \phi_{ij}) \in \Omega(v_i)\} \\ \theta = (\bigcup\limits_{1 \leq i < n} \theta_i) \cup (\pi_\rho, \pi_\rho \neq \mathcal{S})\end{array}}{\Omega \vdash v \leftarrow^{\rho} f(v_1, \ldots, v_n) : \Omega[v \mapsto \theta, \pi_\rho \mapsto (\pi_\rho, \text{true})], \text{true}}$$

Fig. 4. VC Generation rules

the concrete store Γ from the operational semantics. The abstract store Ω maps each program variable to a *guarded value set* θ, consisting of value, constraint pairs. Values $\mathcal{A}$ in the analysis include sources $\mathcal{S}$, constants $\mathcal{C}_1, \ldots, \mathcal{C}_n$, and special variables π which are used to model the unknown return values of library functions. If $(\mathcal{A}_i, \phi_i) \in \Omega(v)$, this means v may be equal to $\mathcal{A}_i$ if constraint ϕ_i is satisfied. *Constraints* ϕ are formed according to the following grammar:

$$\phi := \text{true} \mid \text{false} \mid \mathcal{A} \neq \mathcal{S} \mid f_i \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2$$

Hence, constraints are boolean combinations of flow variables f_i and disequality constraints $\mathcal{A} \neq \mathcal{S}$. As expected, $(\mathcal{S} \neq \mathcal{S}) \equiv \text{false}$ and $(\mathcal{C}_i \neq \mathcal{S}) \equiv \text{true}$ for any $\mathcal{C}_i$. Given an interpretation σ mapping each flow variable to a boolean constant and each π variable to a constant $\mathcal{C}_i$ or $\mathcal{S}$, $\sigma(\phi)$ evaluates to true or false.

Figure 4 presents the analysis using judgments of the form $\Omega \vdash s : \Omega', \phi$. The meaning of this judgment is that, given an abstract store Ω, the analysis of statement s yields a new abstract store Ω' and a verification condition ϕ.

Rules (1) and (2) in Figure 4 describe the analysis of assignments and are straightforward analogues of the concrete semantics. For example, rule (2) for assignments $(v_1 \leftarrow v_2)$ says that if v_2 has guarded value set θ in Ω, then Ω' also maps v_1 to θ. For both rules, the resulting VC is just true since these statements do not contain sinks. Rule (3) for sequencing also closely parallels the concrete semantics. The resulting VC is $\phi_1 \wedge \phi_2$ because the VCs of both s_1 and s_2 must hold to ensure that the program is error-free.

Rule (4) generates the VC for check statements. In this rule, we first retrieve the guarded value set θ of variable v. For each pair, $(\mathcal{A}_i, \phi_i) \in \theta$, we must check that $\mathcal{A}_i$ is not equal to $\mathcal{S}$ under constraint ϕ_i, which is a necessary condition for v to have value $\mathcal{A}_i$. Thus, the generated verification condition is the conjunction of constraints $\phi_i \Rightarrow (\mathcal{A}_i \neq \mathcal{S})$ for each $(\mathcal{A}_i, \phi_i) \in \theta$.

Rule (5) describes the analysis of conditionals. Since the analysis must account for the possibility that either branch may execute, the resulting store Ω' is obtained by taking the *join* of Ω_1 and Ω_2:

Definition 4 *Given two abstract stores Ω_1 and Ω_2, $\Omega_1 \sqcup \Omega_2$ is defined as follows:*

$$(\mathcal{A}, \phi) \in \Omega_1(v) \wedge (\mathcal{A}, \phi') \in \Omega_2(v) \Rightarrow (\mathcal{A}, \phi \vee \phi') \in (\Omega_1 \sqcup \Omega_2)(v)$$
$$(\mathcal{A}, \phi) \in \Omega_i(v) \wedge (\mathcal{A}, \_) \notin \Omega_j(v) \Rightarrow (\mathcal{A}, \phi) \in (\Omega_1 \sqcup \Omega_2)(v)$$

The VC we generate in rule (5) is $\phi_1 \wedge \phi_2$ because the VCs of both branches must hold to guarantee that any execution of the program is error-free.

The last rule (6) describes the analysis of calls to library methods. Since library methods do not contain sinks, the VC here is just true. The more interesting part of this rule is the computation of the new guarded value set θ. To account for the possibility that f's i'th argument may flow to its return value, we introduce a propositional flow variable f_i representing whether or not there is such a flow. Specifically, if $(\mathcal{A}_{ij}, \phi_{ij}) \in \Omega(v_i)$, then $(\mathcal{A}_{ij}, \phi_{ij} \wedge f_i) \in \Omega'(v_i)$, where Ω' is the abstract store after analyzing the function call. Now, in addition, to account for the possibility that f may return a fresh value, v's value set in Ω' also contains a variable π_ρ, which represents an unknown value produced in f. However, since sources are disallowed in library methods, the guard $\pi_\rho \neq \mathcal{S}$ for v in θ stipulates that π_ρ is not a source.

Example 3. Consider the following code snippet:

1. $x \leftarrow \mathcal{S}$; $y \leftarrow \mathcal{C}_2$; if$(\star)$ then $z \leftarrow x$ else $z \leftarrow y$;
2. $a \leftarrow^1 f(z)$; $b \leftarrow^2 g(y)$; $c \leftarrow^3 f(x)$;
3. if$(\star)$ then $d \leftarrow^4 m(a)$ else $d \leftarrow^5 h(b, c)$;
4. check(d)

Here, after line 2, we have:

$$\begin{array}{llll} \Omega(x) & = & \{(\mathcal{S}, \text{true})\} & \Omega(y) = \{(\mathcal{C}_2, \text{true})\} \\ \Omega(z) & = & \{(\mathcal{S}, \text{true}), (\mathcal{C}_2, \text{true})\} & \Omega(a) = \{(\mathcal{S}, f_1), (\mathcal{C}_2, f_1), (\pi_1, \pi_1 \neq \mathcal{S})\} \\ \Omega(b) & = & \{(\mathcal{C}_2, g_1), (\pi_2, \pi_2 \neq \mathcal{S})\} & \Omega(c) = \{(\mathcal{S}, f_1), (\pi_3, \pi_3 \neq \mathcal{S})\} \end{array}$$

Here, f_1, g_1 are flow variables indicating whether f and g's first arguments may flow to their return value. After analyzing the if statement at line 3, we have:

$$(\mathcal{S}, (f_1 \wedge m_1) \vee (f_1 \wedge h_2)) \in \Omega(d)$$

This means that d is tainted at line 4 under constraint $(f_1 \wedge m_1) \vee (f_1 \wedge h_2)$. Here, the first disjunct $(f_1 \wedge m_1)$ comes from the then branch of the if statement, while the second disjunct, $(f_1 \wedge h_2)$ comes from the else branch. Specifically, in the then branch, d is tainted if a is tainted and m's first argument flows to its return value. Since a is tainted under guard f_1, d is tainted under guard $f_1 \wedge m_1$ where m_1 is a flow variable for function m. Similarly, in the else branch, d is tainted if either (i) b is tainted and h's first argument flows to its return value, or (ii) c is tainted and h's second argument flows to its return value. Since $\mathcal{S}$ is not in the value set for b, it is not tainted, and the first condition is false. Since c is tainted under constraint f_1, the second condition is $f_1 \wedge h_2$. Thus, in the else branch, d is tainted under constraint $f_1 \wedge h_2$. Finally, when we take the join of the two value sets for d, we obtain that d is tainted under the guard $(f_1 \wedge m_1) \vee (f_1 \wedge h_2)$ after line 4. Finally, after analyzing the check statement at line 5, the (simplified) VC is given by $((f_1 \wedge m_1) \vee (f_1 \wedge h_2)) \Rightarrow \text{false}$.

3.1 Soundness

The soundness proof of the analysis is provided in Appendix A.

4 Inference of Flow Specifications

Given a program P and its VC ϕ, our goal is now to infer a smallest set candidate flow specifications ψ such that ψ is sufficient to prove the validity of ϕ. At a technical level, we define flow specifications as follows:

Definition 5 *A* flow specification *for a library function f is an assignment from a flow variable f_i to a boolean constant.*

A flow specification for a function f is said to be *correct* if it assigns f_i to false only if there is no possible execution $\sigma = \{s_1, \ldots, s_n\}$ of f where $(\alpha_i, s_1^-) \rightsquigarrow^* (\chi_f, s_n^+)$. In the rest of this section, we assume there is an oracle which can confirm or refute the correctness of a candidate flow specification. This oracle may be a human who can consult the documentation of the library or some other sound analysis capable of analyzing the library code or binary. Since our goal is to minimize the number of queries to the oracle, we are interested in inferring a *minimal specification* sufficient for the verification task.

4.1 Computing Minimal Candidate Flow Specifications

We formulate the minimal specification inference problem in terms of abduction in logic. Specifically, given a VC ϕ for a source-sink problem, and a formula χ representing known flow specifications, we want to infer a formula ψ such that:

$$(1)\ \chi \wedge \psi \models \phi \quad (2)\ \mathrm{SAT}(\chi \wedge \psi)$$

Here, (1) says the candidate specification ψ, together with known specifications χ, should be sufficient to discharge the VC ϕ, and (2) says the candidate specification ψ should not contradict known specifications χ, since such a solution ψ cannot be correct. The inference of a formula ψ satisfying the above requirements is an abduction problem in logic. However, in addition to being a solution to the abduction problem, we require ψ to satisfy two additional requirements:

- First, since ψ represents a set of flow specifications, it should be a conjunction of literals, where each literal is either a flow variable f_i or its negation.
- Second, since we want a *minimal* specification, ψ should contain as few literals as possible. Since each literal in ψ corresponds to a query to the oracle, this minimizes the number of queries needed to verify the client.

Our insight is that we can compute a solution ψ satisfying these two requirements using *minimum-size prime implicants* (MPI) defined as follows:

Definition 6 (Minimum-size prime implicant) *A minimum-size prime implicant (MPI) of a boolean formula φ is a set S of literals such that $\bigwedge_{l_i \in S} l_i \models \varphi$ and for any other set S' such that $|S'| < |S|$, $\bigwedge_{l'_i \in S'} l'_i \not\models \varphi$.*

```
Procedure FindSpec
    input: verification condition φ
    output: inferred specification χ
(1)   χ := true
(2)   while true do
(3)       I := MinPrimeImp(χ ⇒ φ, χ)
(4)       if I = ∅ then return false
(5)       proven := false
(6)       for each l_i ∈ I
(7)           proven := CertifiedByOracle(l_i)
(8)           if proven then χ := χ ∧ l_i
(9)           else (χ := χ ∧ ¬l_i; break)
(10)      if proven then return χ
```

Fig. 5. Algorithm for finding correct flow specifications

Practical algorithms for computing MPIs of boolean formulas have been studied, for example, in [5, 6]. To see how MPIs are useful for solving our abduction problem, observe that the first requirement $\chi \wedge \psi \models \phi$ can be written as $\psi \models \chi \Rightarrow \phi$. Since we want ψ to be a smallest conjunction of literals that implies $\chi \Rightarrow \phi$, a solution to the abduction problem is an MPI of $\chi \Rightarrow \phi$ that does not contradict χ. The algorithm given in [6] can be used to compute an MPI of $\chi \Rightarrow \phi$ consistent with χ, which yields the solution ψ to our abduction problem.

Example 4. Consider again the code from Example 3. Recall that we computed the VC for this program as $\phi : ((f_1 \wedge m_1) \vee (f_1 \wedge h_2)) \Rightarrow \text{false}$. Assuming $\chi = \text{true}$, an MPI for $\chi \Rightarrow \phi$ is $\neg f_1$. Hence, the program is free of source-sink errors if there is no flow from the first argument of f to its return value.

4.2 Computing Correct Minimal Flow Specifications

The solution ψ to the abduction problem from Section 4.1 yields a minimal candidate specification sufficient to verify the client. However, since ψ is effectively a speculation, it does not have to be correct. This section describes a refinement algorithm that interacts with the oracle until a correct specification is found. The FindSpec algorithm used for this purpose is shown in Figure 5. It takes as input the VC ϕ and returns a set of correct flow specifications that are sufficient to discharge the error. If no such specification exists, it returns false.

The idea behind FindSpec is the following: First, at line (3), it computes a set $I = \{l_1, \ldots, l_n\}$ of candidate flow specifications as discussed in Section 4.1 using minimum prime implicants. In the inner loop of FindSpec, we use an oracle to certify each flow specification $l_i \in I$. If the oracle can certify each $l_i \in I$, then we are done. However, if the oracle cannot validate some candidate specification $l_i \in I$, this means I may not be correct, and we therefore backtrack from this choice by breaking out of the inner loop (line 9). In each iteration of the outer while loop, we compute a new candidate specification I using abduction. To ensure that the current solution I is distinct from previous ones, we maintain a formula χ which represents previous answers given by the oracle. This formula χ is initially true, but becomes stronger after every query to the oracle: If the oracle certifies specification l_i, we conjoin l_i with χ; otherwise, we conjoin $\neg l_i$. This strategy ensures that we do not obtain inferences containing literal l_i in

the future. This process continues until we either find a valid proof or conclude that the program cannot be verified relative to the oracle.

Theorem 1. *The FindSpec algorithm is guaranteed to terminate.*

Proof. Given in Appendix B.

Theorem 2. *If there exists a set of correct flow specifications sufficient to discharge ϕ, then FindSpec will not return false assuming completeness of the oracle.*

Proof. Given in Appendix B.

Example 5. Consider again the code from Example 3 and its corresponding VC $\phi : ((f_1 \wedge m_1) \vee (f_1 \wedge h_2)) \Rightarrow$ false. FindSpec first computes an MPI of ϕ, which is $\neg f_1$ as discussed in Example 4. Thus, I starts out as $\{\neg f_1\}$. If the oracle certifies that f's first argument never flows to its return value, the program is verified. Otherwise, we conjoin f_1 with χ (true). In the next iteration, the solution to the abduction problem is $\{\neg m_1, \neg h_2\}$. Now, we ask the oracle to certify $\neg m_1$. If the oracle cannot do so, the algorithm terminates because m_1 is conjoined with χ, and there is no abductive solution consistent with $f_1 \wedge m_1$. If the oracle can certify m_1, the program can be verified iff the oracle can also certify $\neg h_2$.

5 Tracking Flows in the Heap

While the language used in the formalization did not allow pointers, our implementation targets Java, where method calls can introduce flows between any pair of heap objects reachable in the called method. Therefore, to soundly handle flows through the heap, we introduce one flow variable for any ordered pair of abstract memory locations (l_1, l_2) for which the value in l_2 may flow to l_1. That is, a flow variable f_{l_1,l_2} expresses that there is a flow in f from some concrete memory location represented by l_1 to a concrete location represented by l_2. Since Java is type-safe, we consider a flow between two heap locations l_1 and l_2 feasible if the static type of l_1 is a subtype of that of l_2 or vice versa. If these types are class types and field declarations in the library are visible, we also consider their nested fields with their respective types. [4] As an example, consider the following Java code, where library definitions and client code are shown:

```
/* Library Definitions */
class Name { private String name; };
class Phone { private String phone; }
class Contact { private Name n; private Phone p; };

/* Client Code */
String str = TAINTED; Name n = new Name ("John Smith");
Phone p = new Phone(str); Contact c = new Contact(n, p);
Network.send(n);
```

[4] If library declarations are not visible, we conservatively introduce a flow variable for any pair of locations (l_1, l_2) where l_2 is an instance of a library-defined class.

App	LOC	lib calls	possible flows	time	queries	$\frac{\text{queries}}{\text{possible}}$	SSF actual	SSF demonic
ContentProvider	342	26	27	0.9s	2	7.41%	1	3
ContactManager	1017	206	319	5.2s	2	0.63%	1	4
TUIOdroid	1612	437	646	16.4s	2	0.31%	1	24
eu.domob.angulo	1748	141	163	5.0s	1	0.61%	1	22
RemoteDroid	3781	505	680	33.9s	4	0.59%	3	48
tomdroid	10316	1515	2853	126.3s	4	0.14%	2	31
net.rocrail.android	13499	1940	3281	214.0s	4	0.12%	0	103
org.yaaic	18099	1517	2800	96.7s	9	0.32%	1	92
Average	6302	786	1346	62.3s	3.5	0.26%	1.25	40.9

Fig. 6. Experimental Results. The column labeled "SSF actual" shows the number of actual source-sink flows, while "SSF demonic" shows the number of source-sink flows reported when making conservative assumptions about library methods.

Here, assume `TAINTED` is a secret value and `Network.send` is a sink. Since `p.phone` and `str` are type compatible, a boolean variable $Phone_{arg_0,ret.phone}$ expresses a possible flow from the first argument of the `Phone` constructor to the `phone` field of its return value. Similarly, since the `name` and `phone` fields of heap locations `p` and `n` are type compatible, the boolean variables $Contact_{arg_1.phone,arg_0.name}$ and $Contact_{arg_0.name,arg_1.phone}$ express potential flows from `p.phone` to `n.name` and vice versa. Therefore, for this example, our technique generates the VC: $\neg Contact_{arg_1.phone,arg_0.name} \vee \neg Phone_{arg_0,ret.phone}$.

6 Implementation

We have implemented a tool for tracking explicit information flow properties of Android smart phone applications. Android applications are developed in the Java programming language, but make extensive use of the Android software development kit (SDK). Since the Android SDK is many orders of magnitude larger than the typical client application and since it is written in a variety of languages besides Java, it is impractical to perform a precise static analysis of the framework code along with the application code. Therefore, our implementation uses the technique described in this paper to reason about calls to the Android framework when analyzing a given smart phone application.

While the language from Section 2 does not contain many of the standard features of Java, such as pointers and virtual method calls, our implementation handles the full Java language (except some uses of reflection). For reasoning about the heap, we use a flow- and context-sensitive pointer analysis that we introduced in our previous work (see [7]). Since our pointer analysis also does not analyze the Android framework code, we deal with calls to the Android framework methods as outlined in Section 5.

7 Experiments

We used the proposed technique for verifying confidentiality in Android applications. Specifically, we targeted explicit information flow properties where sources correspond to private user data and sinks are methods that send data. Since the proposed technique is not meant for inferring sources and sinks, we manually annotated a total of six different sources and sinks found in these applications. Sources include phone contacts, GPS location, call records, and IMEI number, while sinks include methods that send data over the network and methods for sending SMS messages.

In our experimental evaluation, we chose to focus on Android applications because they are programmed against the complex Android SDK library. While individual Android applications are typically a few thousand or ten thousand lines of code, the size of the entire Android library stack is several millions of lines of code, containing a mix of Java, C, and C++ code. Therefore, Android applications are good examples of software for which it is neither feasible nor desirable to analyze implementations of libraries in order to verify the client.

The eight applications we analyzed range from 342 to 18,099 lines of code and include an instant messaging client, a miniature train controller, an angle measurement software, two remote control programs, a note-taking software, and two small Android developer example applications. With the exception of the miniature train control (net.rocrail.android), all applications contain some source-sink flows. However, many of these flows are necessary for the application to perform its functionality and do not show malicious intent.

The first four columns in Figure 6 give details about the experimental benchmarks. As indicated by the column "lib calls", the number of calls to library methods range from 26 to 1940 calls per application. On average, there are 786 calls to unanalyzed library methods. The column labeled "possible flows" shows the total number of flows that could be introduced due to calls to library methods. On average, there are 1346 possible flows that could arise from library calls.

The next five columns in Figure 6 present analysis results. As shown in the column labeled "time", the analysis takes an average of 62.3 seconds for analyzing an average of 6302 lines of code. The next column labeled "queries" shows the number of queries made to the oracle. As shown in the "queries" column, the number of queries to the oracle range from one to nine, with an average of 3.5 queries across all benchmarks. In our experimental setup, all queries to the oracle were answered by one of the authors who consulted the documentation of the library methods in the Android SDK. Observe that the number of flow specifications that must be confirmed by the user is a very small percentage of the possible flows that could be introduced due to calls to library methods. The column labeled "queries/possible" shows the percentage of flow specifications that must be confirmed by the user relative to all possible flows that could be introduced due to library calls. As the table shows, this percentage is very small; on average, the user only needs to examine 0.26% of all the possible flows.

The next column labeled "SSF actual" in Figure 6 shows the number of source-sink flows identified by our analysis. On these benchmarks, our analysis identifies zero to three possible source-sink flows. We manually inspected all of these flows and found that none of them are spurious. We believe the absence of false positives indicates that the may-flow abstraction is sufficiently precise for summarizing the behavior of libraries for source-sink property verification.

Finally, the last column labeled "SSF demonic" shows the number of source-sink errors identified by the analysis when we make demonic (i.e., conservative) assumptions about flows that could be introduced due to library calls. As shown in Figure 6, there are an average of 40.9 source-sink flows identified by the analysis when we make conservative assumptions about library calls. On aver-

age, this is 33 times larger than the number of actual source-sink flows found in these applications and shows that making conservative assumptions about library method behavior results in a very high number of false alarms.

8 Related Work

Specification Inference Existing work for specification inference [8–10, 4, 11–14] can be classified as static vs. dynamic and library-side vs. client-side. Our technique is client-side and static, but differs from previous work in several ways: First, our technique infers flow specifications for source-sink problems, which is not addressed by existing work. Second, our approach uses a novel form of VC generation and minimum prime implicants to identify the required specifications. Third, our goal is *not* to infer as many facts as possible about the library, but rather to identify exactly those specifications needed to verify a given client.

Specifications for Source-Sink Properties Other work for specification inference, such as [4, 15], also target source-sink problems. In particular, Merlin [4] infers sources, sinks, and sanitizers for explicit information flow problems using probabilistic inference. We believe our proposed technique and [4] are complementary since we assume that sources and sinks are known, whereas [4] does not infer specifications of methods that propagate taint, but that are neither sources nor sinks. The recent work presented in [15] addresses taint analysis of framework-based web applications and gives a specification language for annotating taint-related framework behavior.

Other Source-Sink Checkers Many techniques have been proposed for checking source-sink properties [16–19]. Many of these tools focus on taint analysis in the context of SQL injection attacks [20, 17, 16], where it is important to consider *sanitizers* in addition to sources and sinks. While we have not considered sanitizers in the technical development, our technique can be easily extended to infer flow specifications in the presence of sanitizers. However, we assume that sources, sinks, and sanitizers are known and infer specifications regarding *taint propagation* rather than taint introduction, removal, or consumption.

Use of Abduction in Static Analysis Several other approaches have used abductive inference in the context of program verification [21–23]. Among these, [22] also uses abduction for inferring specifications of unknown procedures but differs from this work in several ways: First, here, our goal is to verify the absence of source-sink errors whereas [22] addresses verifying pointer safety. The second difference is that we consider an algorithmic approach to performing abduction in propositional logic whereas [22, 21] use a rule-based approach in separation logic. Third, while [22] generates a single abductive solution and fails if the candidate specification is wrong, our technique iteratively refines the candidate specification until the program is verified.

Our own previous work uses abductive inference for helping users diagnose warnings generated by static analysis [24]. However, that technique is not useful for inferring flow specifications because the generated constraints do not express possible input-output dependencies due to unknown method calls. Furthermore, the abduction algorithm used here is different: Here, we are interested in conjunc-

tive propositional formulas over flow variables whereas [24] generates possibly disjunctive solutions of Presburger arithmetic formulas.

9 Conclusions and Future Work

We have presented a new technique for source-sink property verification of open programs that call unanalyzed library methods. We have applied the proposed technique to checking confidentiality in Android applications and show that our method can effectively identify the necessary specifications required for ruling out a large number of potential source-sink flows. A promising direction for future research is to combine the proposed technique with dynamic symbolic execution which can rule out many spurious must-not-flow assumptions inferred by our technique.

References

1. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. Model Checking Software (2003) 624–624
2. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: PASTE, ACM (2007) 43–48
3. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: POPL, NY, USA (2002) 1–3
4. Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: Specification inference for explicit information flow problems. In: PLDI. (2009)
5. Silva, J.: On computing minimum size prime implicants. In: International Workshop on Logic Synthesis, Citeseer (1997)
6. Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum satisfying assignments for SMT. In: CAV. (2012)
7. Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. PLDI '11 (2011) 567–577
8. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: ISSTA. (2002) 232–242
9. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. POPL '02 (2002) 4–16
10. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal api rules from imperfect traces. ICSE '06 (2006) 282–291
11. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. POPL '05 (2005) 98–109
12. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. ISSTA '07 (2007) 174–184
13. Beckman, N.E., Nori, A.V.: Probabilistic, modular and scalable inference of typestate specifications. In: PLDI, ACM (2011) 211–221
14. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. PLDI '07 (2007) 123–134
15. Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4F: taint analysis of framework-based web applications. OOPSLA '11 (2011) 1053–1068
16. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. PLDI '09 (2009) 87–97
17. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security Symposium. SSYM'05 (2005) 18–18

18. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. OSDI'10 (2010) 1–6
19. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. ISSTA '07 (2007) 196–206
20. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI, ACM (2007) 32–41
21. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL (2009) 289–300
22. Luo, C., Craciun, F., Qin, S., He, G., Chin, W.N.: Verifying pointer safety for programs with unknown calls. J. Symb. Comput. **45**(11) (2010) 1163–1183
23. Giacobazzi, R.: Abductive analysis of modular logic programs. In: International Symposium on Logic programming, Citeseer (1994) 377–391
24. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI. (2012)

Appendix A: Soundness of Analysis

In this section, we prove the soundness of the analysis presented in Section 3.

First, to allow us to relate the concrete and abstract semantics in a precise way, we introduce some auxiliary concepts that are necessary for stating the soundness theorem.

Definition 7 (Flow environment $\mathcal{F}$) *Given a concrete execution $\sigma = s_1 s_2, \ldots, s_k$ of a program P, $\mathcal{F}_\sigma$ is an environment that maps each flow variable f_i to a boolean value. Specifically, $\mathcal{F}_\sigma(f_i) = \text{true}$ iff $(\alpha_i, s_i^+) \rightsquigarrow^* (\chi_f, s_j^-)$ for some $s_i, s_j \in \sigma$ such that $s_i = \text{call}^\rho f(\ldots)$ and $s_j = \text{return}^\rho f$.*

In other words, flow environment $\mathcal{F}_\sigma$ tracks whether there is a flow from f's i'th argument to its return value in a concrete execution σ. It is easy to instrument the concrete semantics in a way that allows the computation of $\mathcal{F}_\sigma$.

Next, we describe how to evaluate the terms and constraints used in the analysis under concrete store Γ and flow environment $\mathcal{F}$:

Definition 8 (Evaluation of terms) $\text{eval}(t, \Gamma)$ *evaluates term t to a constant under environment Γ:*

$$\begin{array}{l} \text{eval}(c, \Gamma) = c \qquad (c \in \{\mathcal{S}, \mathcal{C}_1, \ldots, \mathcal{C}_k\}) \\ \text{eval}(v, \Gamma) = \Gamma(v) \ (v \text{ a variable}) \end{array}$$

Definition 9 (Evaluation of constraint) *For a concrete execution with flow environment $\mathcal{F}$, $\text{eval}_\mathcal{F}(\phi, \Gamma)$ evaluates constraint ϕ to a boolean constant as follows:*

$$\begin{array}{r} \text{eval}_\mathcal{F}(f_i, \Gamma) = \mathcal{F}(f_i) \\ \text{eval}_\mathcal{F}(t_1 \neq t_2, \Gamma) = \text{eval}(t_1, \Gamma) \neq \text{eval}(t_2, \Gamma) \\ \text{eval}_\mathcal{F}(\phi_1 \circ \phi_2, \Gamma) = \text{eval}(\phi_1, \Gamma) \circ \text{eval}(\phi_2, \Gamma) \quad (\circ \in \{\wedge, \vee, \Rightarrow\}) \end{array}$$

Definition 10 (Evaluation of θ) *$\mathcal{F}$, $\text{eval}_\mathcal{F}(\theta, \Gamma)$ evaluates guarded value set θ as follows:*

$$\mathit{eval}_\mathcal{F}(\theta, \Gamma) = \left\{ (c_i, b_i) \;\middle|\; \begin{array}{l} c_i = \text{eval}(t_i, \Gamma) \wedge \\ b_i = \text{eval}_\mathcal{F}(\phi_i, \Gamma) \wedge (t_i, \phi_i) \in \theta \end{array} \right\}$$

Definition 11 (Evaluation of Ω) *For a concrete execution with flow environment $\mathcal{F}$, $\text{eval}_\mathcal{F}(\Omega, \Gamma)$ evaluates abstract environment Ω under Γ. Specifically, for every $v \in dom(\Omega)$, we have:*

$$\text{eval}_\mathcal{F}(\Omega, \Gamma) = [v \mapsto \theta \mid v \in \Omega \quad \theta = \text{eval}_\mathcal{F}(\Omega(v), \Gamma))]$$

Given these definitions, we can now state *agreement* between concrete and abstract states:

Definition 12 (Agreement) *Given a concrete execution with flow environment $\mathcal{F}$, we say that a concrete environment Γ agrees with abstract environment Ω, written $\Gamma \sim_\mathcal{F} \Omega$, if and only if the following conditions hold:*

1. *If* $v \in dom(\Gamma)$*, then* $v \in dom(\Omega)$
2. $\forall v \in \mathrm{dom}(\Gamma)$*, if* $\Gamma(v) = c$ *and* $\Omega(v) = \theta$*, then:*

$$(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma)$$

The following theorem states the soundness of the analysis:

Theorem 3 (Soundness). *Let* σ *be an execution of program* P *such that* $\Pi, \Gamma \vdash P : \Gamma', b$ *according to the concrete semantics, and let* $\mathcal{F}$ *be the flow environment for* σ*. Suppose* $\Omega \vdash P : \Omega', \phi$ *according to the abstract semantics. If* $\Gamma \sim_{\mathcal{F}} \Omega$*, then* $\Gamma' \sim_{\mathcal{F}} \Omega'$ *and* $\mathrm{eval}_{\mathcal{F}}(\phi, \Gamma') \Rightarrow b$*.*

Proof. The proof is by structural induction.

- Let $P = (v \leftarrow c)$. Since $\Gamma \sim_{\mathcal{F}} \Omega$, $\forall x \neq v$. $\Gamma'(x) \sim_{\mathcal{F}} \Omega'(x)$. Furthermore, since $\Gamma'(v) = c$ and $\Omega'(v) = (c, \mathrm{true})$, we have $\Gamma'(v) \sim_{\mathcal{F}} \Omega'(v)$. Thus, $\Gamma' \sim_{\mathcal{F}} \Omega'$. Finally, since $\phi = \mathrm{true}$ and $b = \mathrm{true}$, $eval_{\mathcal{F}}(\phi, \Gamma') \Rightarrow b$.
- Let $P = (v_1 \leftarrow v_2)$. Since $\Gamma(v_2) = c$ and $\Omega(v_2) = \theta$, we have $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma)$ because $\Gamma \sim_{\mathcal{F}} \Omega$. Hence, $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\Omega'(v_1), \Gamma')$; thus $\Omega' \sim_{\mathcal{F}} \Gamma'$. Finally, since $\phi = \mathrm{true}$ and $b = \mathrm{true}$, $eval_{\mathcal{F}}(\phi, \Gamma') \Rightarrow b$.
- Let $P = s_1; s_2$. By the inductive hypothesis, $\Omega_1 \sim_{\mathcal{F}} \Gamma_1$. Thus, also by the inductive hypothesis, $\Omega_2 \sim_{\mathcal{F}} \Gamma_2$. Also, by the inductive hypothesis, we have $eval_{\mathcal{F}}(\phi_1, \Gamma_1) \Rightarrow b_1$ and $eval_{\mathcal{F}}(\phi_2, \Gamma_2) \Rightarrow b_2$. Note that Γ_2 must agree with Γ_1 on the values of all π_ρ variables used in the constraints, thus $eval_{\mathcal{F}}(\phi_1, \Gamma_2) = eval_{\mathcal{F}}(\phi_1, \Gamma_1)$. Finally, be definition of $eval_{\mathcal{F}}$, $eval_{\mathcal{F}}(\phi_1 \wedge \phi_2, \Gamma_2) = eval(\phi_1, \Gamma_2) \wedge eval_{\mathcal{F}}(\phi_2, \Gamma_2)$. Hence, $eval_{\mathcal{F}}(\phi_1 \wedge \phi_2, \Gamma_2) \Rightarrow b_1 \wedge b_2$.
- $P = \mathrm{check}(v)$. The interesting case is when $\Gamma(v) = \mathcal{S}$. Since $\Gamma \sim_{\mathcal{F}} \Omega$, $(\mathcal{S}, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma)$. Thus, there must exist some $(\mathcal{A}_i, \phi_i) \in \theta$ such that $eval_{\mathcal{F}}(\mathcal{A}_i, \Gamma) = \mathcal{S}$ and $eval_{\mathcal{F}}(\phi_i, \Gamma) = \mathrm{true}$. Since $\phi = \bigwedge_{(\mathcal{A}_i, \phi_i) \in \theta} \phi_i \Rightarrow (\mathcal{A} \neq \mathcal{S})$, $eval_{\mathcal{F}}(\phi, \Gamma) = \mathrm{false}$.
- $P = \mathrm{if}(\star)$ then s_1 else s_2. Without loss of generality, suppose $(s_1 \oplus s_2) = s_1$. By the inductive hypothesis, $\Gamma_1 \sim_{\mathcal{F}} \Omega_1$. By definition of $\sqcup$, if $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma_1)$, then $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta \sqcup \theta', \Gamma_1)$ for any θ'. Thus, $\Gamma_1 \sim_{\mathcal{F}} \Omega_1 \sqcup \Omega_2$. We also have that $eval_{\mathcal{F}}(\phi_1 \wedge \phi_2, \Gamma_1) \Rightarrow b$ because $eval_{\mathcal{F}}(\phi_1 \wedge \phi_2, \Gamma_1) = eval_{\mathcal{F}}(\phi_1, \Gamma_1) \wedge eval_{\mathcal{F}}(\phi_2, \Gamma_2)$, and $eval_{\mathcal{F}}(\phi_1, \Gamma_1) \Rightarrow b_1$ by the inductive hypothesis.
- $P = v \leftarrow^{\rho} f(v_1, \ldots, v_n)$. There are two cases to consider:
  Case 1: Suppose $(v_i, P^-) \rightsquigarrow^* (v, P^+)$ for some v_i (where P^- denotes the control point before call$^{\rho} f(\ldots)$ instruction, and P^+ denotes the control point right after the return return$^{\rho} f$ instruction). Since $\Gamma'(v) = c$, this implies $\Gamma(v_i) = c$ because $(v_i, P^-) \rightsquigarrow^* (v, P^+)$. Since $\Gamma \sim_{\mathcal{F}} \Omega$, we have $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\Omega(v_i), \Gamma)$. Since $(v_i, P^-) \rightsquigarrow^* (v, P^+)$, by definition of $\mathcal{F}$, $\mathcal{F}(f_i) = \mathrm{true}$. Therefore, $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma)$. Hence, $\Gamma' \sim_{\mathcal{F}} \Omega'$.
  Case 2: Suppose $(v_i, P^-) \not\rightsquigarrow^* (v, P^+)$ for any v_i. This implies $\Gamma'(v) = c \neq \mathcal{S}$ since the constant $\mathcal{S}$ is disallowed in f. By construction of θ in Ω', $(c, \mathrm{true}) \in eval_{\mathcal{F}}(\theta, \Gamma')$ since $(\pi_\rho, \pi_\rho \neq S) \in \mathcal{S}$ and $\Gamma'(\pi_\rho) = c$ and $c \neq \mathcal{S}$.

Effectively, the soundness theorem states that the verification condition computed by our analysis is a safe overapproximation of program correctness. The following corollary follows immediately from Theorem 3:

Corollary 1. *Let P be a program such that $\Omega \vdash P : \Omega', \phi$. If ϕ is valid (i.e., $\models \phi$), then P does not have a source-sink error in any execution.*

Appendix B: Termination and Completeness of FindSpec

Theorem 2. (Termination) FindSpec algorithm is guaranteed to terminate.

Proof. First, observe that ϕ contains a finite number of flow variables, therefore there are a finite number of *distinct* flow specification queries that can be asked to the oracle. Second, if we make a query l_i to the oracle, no future query q will be equivalent to l_i or its negation. To see this, consider two cases: (i) If l_i is certified by the oracle, $\chi = l_i \wedge \varphi$ for some formula φ. Now, $q \not\equiv \neg l_i$ because each literal in I must be consistent with χ. Furthermore, we also have $q \not\equiv l_i$ because I containing l_i cannot be a *minimum* prime implicant of $\chi \Rightarrow \phi$ because $I \backslash l_i$ is also a prime implicant of $\chi \Rightarrow \phi$ for any conjunct χ containing l_i. (ii) The proof when l_i cannot be certified by the oracle is analogous.

Theorem 3. (Relative Completeness) If there exists a set of correct flow specifications sufficient to discharge ϕ, then FindSpec will not return false assuming completeness of the oracle.

Proof. Suppose there exists a specification I sufficient to discharge ϕ (i.e., $I \models \phi$), but FindSpec returns false. This means that for every χ at line (3) of the algorithm, either (i) $I \not\models \chi \Rightarrow \phi$ or (ii) I and χ are not consistent. Here, (i) is not possible because if $I \models \phi$, then we also have $I \models \chi \Rightarrow \phi$. For (ii), observe that, for $I \wedge \chi$ to be unsatisfiable, I and χ must disagree on the assignment to at least one flow variable. But since the oracle is complete, this is not possible if I is indeed a correct specification.