Leveraging Historical Versions of Android Apps for Efficient and Precise Taint Analysis

John Jenkins
Washington State University, Pullman
john.jenkins@wsu.edu

Haipeng Cai
Washington State University, Pullman
hcai@eecs.wsu.edu

ABSTRACT

Today, computing on various Android devices is pervasive. However, growing security vulnerabilities and attacks in the Android ecosystem constitute various threats through user apps. Taint analysis is a common technique for defending against these threats, yet it suffers from challenges in attaining practical simultaneous scalability and effectiveness. This paper presents a novel approach to fast and precise taint checking, called *incremental taint analysis*, by exploiting the evolving nature of Android apps. The analysis narrows down the search space of taint checking from an entire app, as conventionally addressed, to the parts of the program that are different from its previous versions. This technique improves the overall efficiency of checking multiple versions of the app as it evolves. We have implemented the techniques as a tool prototype, EvoTaint, and evaluated our analysis by applying it to real-world evolving Android apps. Our preliminary results show that the incremental approach largely reduced the cost of taint analysis, by 78.6% on average, yet without sacrificing the analysis effectiveness, relative to a representative precise taint analysis as the baseline.

KEYWORDS

Android, taint analysis, incremental, evolution, differencing

1 INTRODUCTION

One common approach to defending application security in Android is taint checking [9], a technique that examines a given app against possible exposure or leakage to external, unauthorized (even malicious) parties of security-sensitive and/or private data of the app user. To make its results affordable to inspect in terms of time cost, taint analysis needs to be precise, reporting as few false alerts as possible [15]. To minimize the risk of missing actual threats, the analysis needs to also produce complete (or called *safe*) results—capturing as many true security risks as possible [11]. An *accurate* taint analysis would both be precise and report security threats completely. However, an accurate (effective) taint analysis is known to be highly expensive [2, 11], as the underlying rationale for identifying false alerts hence removing them while producing safe results typically involves costly modeling and reasoning about code semantics and app behaviors. For example, a representative taint analysis for Android, FlowDroid [2], can often take a few hours to check a single app when working at its best-precision setting, while settings for better efficiency could result in a significant drop in its precision. As it stands, many existing solutions suffer from efficiency issues hence practicality challenges—they are too heavyweight thus unable to scale to large, real-world apps. Other existing approaches take rapid yet rough approaches hence producing too many false alerts. These consistently occurring false alerts may ultimately impede the adoptability of the technique as the users (e.g., security analysts) may not be able to afford the prohibitive cost of inspecting the large (imprecise) results.

To overcome these challenges, we propose a novel security analysis called *incremental taint analysis*. Like software applications in other domains, Android apps, malicious or benign, constantly evolve as driven by incremental changes [18, 21]. However, existing taint analyses treat each version of an app during its evolution as an independent program without considering the incremental nature of app evolution, thus incurring excessive overall costs for running the analysis on the app during its evolution. The proposed incremental taint analysis exploits the fact that incremental changes are generally small in scale, which are potentially much cheaper to analyze relative to analyzing the entire program. Given multiple evolving versions of an app, our analysis starts with a whole-program taint analysis on the first version (i.e, *base version*). For each evolved version of the app, the changes relative to the base version are computed and the affected program entities are identified through impact analysis. New taint flows due to the change impact are then synthesized with those in the base version to produce the final analysis results of the evolved app.

To assess the cost and effectiveness of our approach, we have implemented the incremental taint analysis as an open-source tool, EvoTaint (Taint analysis of Evolving apps), and applied it to 19 real-world benchmarks of varied sizes and application domains that are chosen from Google Play based on their popularity. For each benchmark, we obtained two historical versions that have varying time gaps, and compared the analysis cost of EvoTaint versus FlowDroid as a precise baseline technique on the evolved versions. Our preliminary results on this particular benchmark suite revealed significant cost-effectiveness improvement over the baseline achieved by our analysis. On average, EvoTaint reduced the analysis cost by 78.6% on average while not penalizing the analysis accuracy at all. Intuitively, the cost reduction is even greater when more evolving versions of an app are involved in the incremental analysis. Given the considerable number of evolving versions on large app stores like Google Play, our technique provides a promising solution to fast and precise screening of apps against security vulnerabilities.

The main contributions of this paper are:

- We proposed a novel approach, *incremental taint analysis*, to checking evolving Android apps against security vulnerabilities, which leverages historical versions of apps to improve the cost-effectiveness of taint analysis.

MSR '18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196433

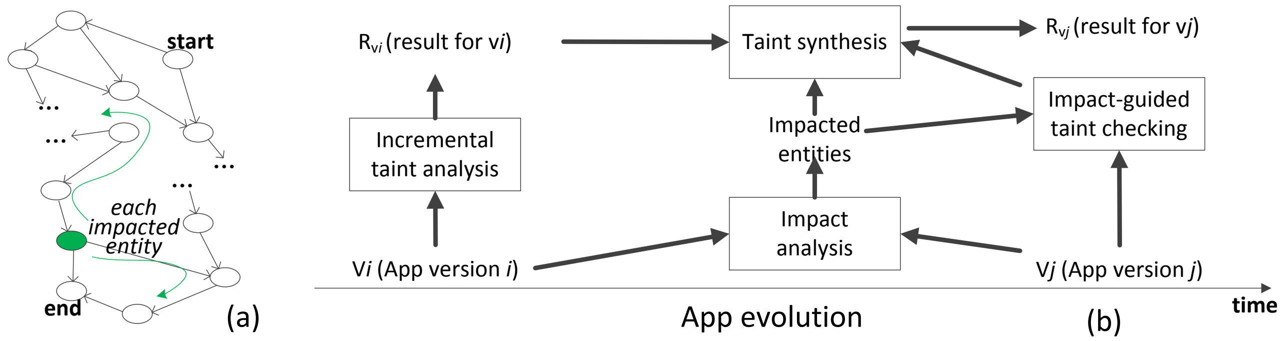

Figure 1: An overview of our incremental approach to taint analysis. It examines only the parts impacted by the incremental changes (from V_i to V_j, $i < j$) to compute taint flows to sources/sinks from each impacted entity (a), and synthesizes the solution for V_i with the impact-guided taint checking results to obtain the solution for V_j (b).

- We developed a new, open-source tool, EvoTaint, which implements the incremental taint analysis and evaluated it against real-world evolving Android apps, and demonstrated the promising benefits of the proposed approach.

2 APPROACH

Our approach aims at a significant reduction of the time cost of a accurate (yet costly) taint analysis, so as to provide a solution that is both efficient and effective for practical adoption. To that end, our analysis focuses on the changes between different versions of an evolving app and on the parts of the newer version that are affected by the changes, instead of analyzing each version as an independent app. In this way, the analysis cost will be amortized across multiple versions of an app. We first give an overview of our approach and then describe the key technical components separately.

2.1 Overview

Figure 1 depicts our incremental approach to taint analysis. In particular, the key idea of the approach is illustrated in Figure 1 (a). Multiple versions of an evolving Android app under vetting are considered and sensitive information flows are examined on the basis of the analysis results of previous versions. Analysis of evolved versions focus on program entities affected by the incremental code changes between two versions of the same app to save analysis cost, rather than exhaustively examining all possible paths between any source and any sink in the evolved app.

The high-level process flow of our technique is illustrated in Figure 1 (b). Given an earlier version V_i and a later (evolved) version V_j of an app, a change *impact analysis* is first performed to compute the set (i.e., *impact set*) of program entities that are impacted by the changes introduced in V_j relative to V_i. Then, taint flows induced by the impact set are computed through an *impact-guided taint checking* algorithm. The analysis result R_{vi} for V_i is obtained either through a whole-program (i.e., conventional) taint analysis if it is the base version; otherwise, without loss of generality, R_{vi} is obtained through incremental taint analysis (recursively). Finally, the impact-guided analysis result is *synthesized* with R_{vi} to derive the full set of taint flows in V_j.

2.2 Analysis of the Original Version

Each pass of incremental taint analysis addresses two versions of an app: the *original* version (e.g., V_i) before changes are made, and the *changed/evolved* version (e.g., V_j) that incorporated the changes. As mentioned above, taint analysis of the original version is performed incrementally relative to an even earlier version of the app as the base version, if available. When the original itself is a base version, we perform the analysis with the conventional approach: the core is an exhaustive, pair-wise reachability analysis that tracks all possible sensitive data flows between all predefined sources and sinks. In particular for Android apps in this work, we use FlowDroid [2] for the conventional whole-program taint analysis. For precise analysis results, we adopt the default, conservative settings for flow and context sensitivity.

2.3 Impact Analysis

The key to the efficiency enhancement with incremental taint analysis is to avoid the redundant computation for program entities of the original app that are not affected by the changes during the evolution. We thus perform a dependence-based change impact analysis [5] to compute the impacted entities. Further, the impact analysis is performed at method level [7], which trades precision for efficiency. In consequence, false positives in the impact set (i.e., methods that are actually not impacted but reported so) lead to extra costs in the impact-guided taint checking (which could be avoided through a more precise impact analysis). However, our method-level abstraction does not sacrifice the safety of the resulting impact set, thus the accuracy of the taint checking is not affected. Also, the extra costs can be counteracted (at least partially) by the cost savings in impact analysis (resulted from the method-level approximation).

More specifically for Android apps, our impact analysis first computes the code changes through app differencing between the original and evolved app versions, and then computes the impact set of the changes. To obtain the method-level code changes (i.e., changed methods), all methods of both app versions are traversed and contrasted, resulting in the lists of methods added, modified, and deleted during the evolution (i.e., the *change set*). For the impact-set computation step, we exploit our method-level impact analysis [7], which builds the method dependence graph [6] and obtains the impact set through (method-level) forward slicing on the graph. Specifically, the impact sets of modified and deleted methods are computed from the original app while the impact sets of the added methods have to be computed from the evolved app (since the original app does not include those methods). Accordingly, the impact analysis produces the impact sets of the three types of changes

ALGORITHM 1: Impact-Guided Taint Checking

```
Input: $V_{org}$ - original app, $V_{evo}$ - evolved app
Output: $R$ - map from change type to affected taint flows
1  let $\mathcal{I}$ be the map from change type to impact set for $V_{org}$ and $V_{evo}$
2  let $G_{org}$ be the call graph of $V_{org}$
3  let $G_{evo}$ be the call graph of $V_{evo}$
4  foreach change type $t \in \{A, M, D\}$ do
5      foreach method $m \in \mathcal{I}[t]$ do
6          $G = G_{org}$
7          if $t$==$A$ then
8              $G = G_{evo}$
9          end
10         identify the set $S$ of sources backward-reachable from $m$ on $G$
11         identify the set $T$ of sinks forward-reachable from $m$ on $G$
12         $S[t]$=$S[t] \cup S$
13         $T[t]$=$T[t] \cup T$
14     end
15     $R[t]$ = getDataFlowPaths($S[t]$, $T[t]$, $V_{evo}$)
16 end
```

separately: addition (A), modification (M), and deletion (D). The changed methods themselves are also considered impacted.

2.4 Impact-Guided Taint Checking

With the change impact set obtained, our incremental taint checking aims to compute the influence of the changes on the taint analysis result of the original app version. Since precise taint checking addresses the (data-flow) reachability from sources to sinks that are specified beforehand, the key to our impact-guided taint analysis lies in reducing the scope of sources and sinks by identifying those that are (indirectly) affected by the changes.

Algorithm 1 shows the impact-guided taint checking algorithm in pseudo code. The algorithm takes the two app versions V_{org} and V_{evo} and computes the taint flows associated with the three types of changes. The change type is used to retrieve the corresponding impact set from the results of impact analysis as described above (line 1). The call graph for both app versions are constructed (lines 2–3) for identifying affected sources and sinks in the following loop (lines 4–16), through backward and forward traversal (lines 10–11) on the appropriate call graph, respectively. In particular, for impact sets of M and D changes, the call graph of V_{org} is used for the reachability analysis (line 6), while for impact sets of A changes, the call graph of V_{evo} is used (lines 7–9) since those impacted methods (e.g., the methods added to V_{evo}) are not all included in V_{org}. After the sources and sinks affected by changes of type t, the algorithm computes the data-flow reachability from any of the sources to any of the sinks in V_{evo}, by invoking a subroutine getDataFlowPaths (line 15) which essentially performs the conventional taint analysis with respect to the given lists of sources and sinks. The algorithm produces the taint flow paths associated with the three change types as the output.

2.5 Taint Synthesis

The taint analysis result for the evolved app V_{evo} is obtained by synthesizing the result R_{org} for the base version V_{org} with the result R of the impact-guided taint checking (i.e., output of Algorithm 1). Specifically, R_{org} = getDataFlowPaths(S_{full}, T_{full}, V_{org}), where S_{full} and T_{full} are the full lists of all possible sources and sinks considered, respectively. Then taint synthesis works in three steps. First, flow paths in R_{org} that pass through any method in the three impact sets are removed from R_{org}. Second, flows paths in $R[D]$ that pass through any deleted method in the change set are removed from $R[D]$. Third, remaining flow paths in R_{org} and $R[D]$ are merged with all paths in $R[A]$ and $R[D]$ to produce the full set of taint flow paths in V_{evo}.

2.6 Implementation and Limitations

We implemented our technique as a tool, EvoTaint, based on Soot [13] and FlowDroid [2]. In particular, the conventional taint analysis routine (getDataFlowPaths) is immediately built on FlowDroid. We also preceded our impact analysis with a preprocessing step which deobfuscates all apps under analysis. Since many real-world Android apps are obfuscated, which impedes both the app differencing and impact-guided taint checking in our technique, this is a necessary engineering step in a practical tool. To deobfuscate an app, we used two state-of-the-art Android deobfuscators, Simplify [10] and DeGuard [4] (as the primary and secondary options, respectively). Thus, our current implementation will not work on apps that either cannot be analyzed by Soot/FlowDroid or cannot be deobfuscated with the two underlying tools (a more advanced deobfuscator can be plugged in to replace them). EvoTaint has been made available as an open-source project at

https://bitbucket.org/sabatu/iterative-taint-analysis

Our technique currently achieves efficiency enhancement in the incremental taint analysis primarily by reducing the sources and sinks to be applied to a conventional taint analysis (line 15 of Algorithm 1). Directly computing affected taint flows via data flow analysis would avoid invoking the getDataFlowPaths routine, which may further speed up the incremental taint analysis. Also, it should be noted that the cost-effectiveness of EvoTaint is constrained by the number of flow paths affected by the changes between the versions under analysis. In the case of this number being very large, the cost of incremental analysis may not be paid off.

3 EVALUATION

Experimental setup. For evaluating our incremental taint analysis, we applied EvoTaint to 19 popular Android apps obtained from the Google Play store. For each of these benchmarks, we downloaded two historical versions, and treated the earlier version as the base/original app while the later version serves as the evolved app. We aimed to answer the following two questions: (1) does our proposed analysis achieve better efficiency over conventional taint analysis and, if it does, by how much? (2) does our analysis achieve the same accuracy in taint checking as the conventional approach?

Accordingly, we measured the time cost of EvoTaint versus FlowDroid as the baseline on the evolved version of each benchmark, since both approaches share the same analysis cost on the base versions. On the evolved versions, FlowDroid treats each as an independent app and performs a whole-program taint analysis, while EvoTaint performs the incremental taint analysis proposed. Thus, we only compared the costs for the evolved versions. More specifically, we computed the percentage cost reduction achieved by EvoTaint relative to the cost of FlowDroid on each benchmark. For the same reason, we only compared the resulting taint flow paths in the evolved versions as produced by the two tools.

Efficiency improvement. Figure 2 depicts the percentages of cost reduction obtained by our approach on the 19 benchmarks (listed on the x axis). The height of each bar in the chart represents the

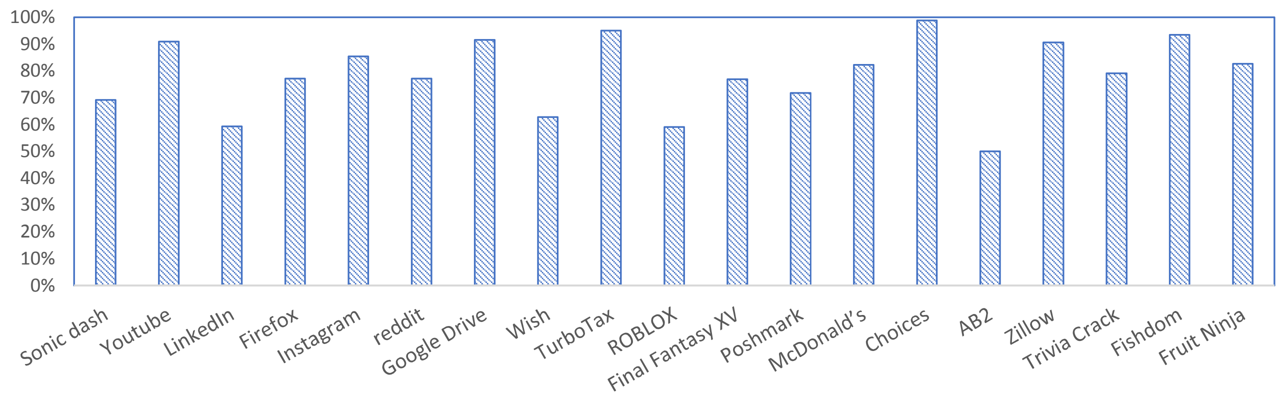

Figure 2: Cost reduction (y axis) achieved by our incremental taint analysis with EvoTaint relative to the conventional approach with FlowDroid on the evolved versions of our benchmarks (x axis).

percentage of total analysis cost on the evolved version by which EvoTaint reduced over the baseline. For instance, on the `Trivia Crack` app, EvoTaint reduced the cost of conventional taint analysis by about 80%. EvoTaint costs included those for deobfuscation.

Overall, the cost reduction ranged from 50% to over 98%, for a mean efficiency improvement percentage of 78.6%—on average, EvoTaint only took 20% of the cost incurred by the baseline. In many individual cases (e.g., `Youtube`, `Google Drive`, `TurboTax`, `Zillow`, `Fishdom`, and `Choices`), our incremental analysis drastically improved the baseline analysis efficiency by over 90%. These numbers revealed that our incremental approach can greatly reduce the cost of taint analysis for Android apps by leveraging the incremental nature of code changes during app evolution.

Effectiveness (accuracy). Given that our goal is to improve the efficiency of taint analysis without sacrificing the analysis effectiveness, it is essential that the taint analysis results produced by our approach were as accurate as the results produced by the baseline approach. We thus checked the taint flow paths as the outputs of EvoTaint versus FlowDroid on the evolved version of each benchmark. For each of the 19 apps, EvoTaint produced the same set of paths as FlowDroid did. Thus, both tools achieved the same accuracy of taint checking consistently on our benchmark suite.

In sum, our preliminary results show that the incremental taint analysis we proposed improved the efficiency of taint checking over the conventional, whole-program analysis approach, without sacrificing analysis effectiveness. Therefore, our approach appeared to have considerably higher *cost-effectiveness* than the conventional taint analysis for Android apps whose earlier versions are available and have been analyzed before.

4 RELATED WORK

Historically, incremental information-flow analysis has been demonstrated or employed to address efficiency challenges in a variety of software engineering tasks, such as data flow computation [17], program testing [3], parallelization [19], change-impact analysis [12, 14], and maintenance tasks [16]. Yet, it has not been as widely used for security defense in general [20], and has not yet been utilized for *code-based* security analysis such as taint checking, a particular form of information flow analysis. Given the large numbers of sources and sinks involved, conventional approaches [2, 11] to taint analysis can suffer from excessive costs when analyzing evolving apps as they do not consider/exploit the incremental nature of app evolution. In contrast, our incremental approach to taint checking takes advantage of that nature to drastically improve the efficiency of precise taint analysis.

Reviser [1] updates interprocedural data-flow facts for incremental code changes, yet it remains unclear how it can be utilized for speeding up taint analysis for evolving Android apps. On the other hand, incorporating the incremental data-flow analysis in our approach could potentially further enhance its efficiency while maintaining the taint-checking accuracy.

Cheetah [8] aims to decrease the time needed for taint analysis of Android apps, a goal shared by our incremental taint analysis. Cheetah took a different approach based on just-in-time static analysis, which leverages the current development context (e.g., code edit points) to compute results of different complexity with varied priority in an integrated development environment. The goal of Cheetah is to reduce the interruption introduced by the use of a static analysis tool during code development through an interactive paradigm of program analysis. Thus, Cheetah suits the scenarios in which developers would like to be notified of vulnerable information flows in an app being developed during the coding process. In contrast, our incremental taint analysis offers a solution to quickly discovering vulnerable information flows in finished apps during their evolutionary production.

5 CONCLUSION

We have presented a novel approach to performing taint analysis of evolving apps, called incremental taint analysis, for enhanced cost-effectiveness over the conventional, whole-program analysis approach. Our technique leverages the historical versions of Android apps and the taint analysis results on them to largely reduce the cost of taint checking for the evolved versions, by avoiding recomputation for code entities that are not changed nor impacted by the changes. Our preliminary results on 19 real-world popularly used evolving Android apps show that our approach can reduce the conventional precise taint analysis cost by 78.6% on average and in many cases up to over 90%, yet without sacrificing the analysis accuracy. As a result, our approach offers significant cost-effectiveness for vetting Android apps during their evolution. We have made available online EvoTaint, our open-source tool implementation of the incremental taint analysis. As future work, we plan to extend our analysis to apps that share similar code-flow paths, but originate from completely dissimilar repositories.

REFERENCES

[1] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*.

[3] Samuel Bates and Susan Horwitz. 1993. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 384–396.

[4] Benjamin Bichsel, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2017. Statistical Deobfuscation for Android. http://apk-deguard.com/. (2017).

[5] Haipeng Cai. 2015. *Cost-effective dependency analysis for reliable software evolution*. University of Notre Dame.

[6] Haipeng Cai and Raul Santelices. 2015. Abstracting program dependencies using the method dependence graph. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 49–58.

[7] Haipeng Cai and Raul Santelices. 2016. Method-Level Program Dependence Abstraction and Its Application to Impact Analysis. *Journal of Systems and Software (JSS)* 122 (2016), 311–326.

[8] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Cheetah: just-in-time taint analysis for android apps. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. 39–42.

[9] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2015. Android security: a survey of issues, malware penetration, and defenses. *Communications Surveys & Tutorials, IEEE* 17, 2 (2015), 998–1022.

[10] Caleb Fenton. 2018. Generic Android Deobfuscator. https://github.com/CalebFenton/simplify. (2018).

[11] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilhamy, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*.

[12] Shengjian Guo, Markus Kusano, and Chao Wang. 2016. Conc-iSE: Incremental symbolic execution of concurrent software. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 531–542.

[13] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*.

[14] James Law and Gregg Rothermel. 2003. Incremental dynamic impact analysis for evolving software systems. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 430–441.

[15] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting.. In *NDSS*.

[16] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 504–515.

[17] Lori L Pollock and Mary Lou Soffa. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering* 15, 12 (1989), 1537–1549.

[18] Vaclav Rajlich and Prashant Gosavi. 2004. Incremental change in object-oriented programming. *IEEE Software* 21, 4 (2004), 62–69.

[19] Kevin Smith, Bill Appelbe, and Kurt Stirewalt. 1990. Incremental Dependence Analysis for Interactive Parallelization. In *Proceedings of the 4th International Conference on Supercomputing*. 330–341.

[20] Basie Von Solms and Rossouw Von Solms. 2001. Incremental information security certification. *Computers & Security* 20, 4 (2001), 308–310.

[21] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. of the IEEE Symposium on Security and Privacy*. 95–109.