# SEALANT: A Detection and Visualization Tool for Inter-App Security Vulnerabilities in Android

Youn Kyu Lee, Peera Yoodee, Arman Shahbazian, Daye Nam, and Nenad Medvidovic
Computer Science Department, University of Southern California
941 Bloom Walk, Los Angeles, California, USA 90089
{younkyul, yoodee, armansha, dayenam, neno}@usc.edu

***Abstract*—Android's flexible communication model allows interactions among third-party apps, but it also leads to inter-app security vulnerabilities. Specifically, malicious apps can eavesdrop on interactions between other apps or exploit the functionality of those apps, which can expose a user's sensitive information to attackers. While the state-of-the-art tools have focused on detecting inter-app vulnerabilities in Android, they neither accurately analyze realistically large numbers of apps nor effectively deliver the identified issues to users. This paper presents *SEALANT*, a novel tool that combines static analysis and visualization techniques that, together, enable accurate identification of inter-app vulnerabilities as well as their systematic visualization. *SEALANT* statically analyzes architectural information of a given set of apps, infers vulnerable communication channels where inter-app attacks can be launched, and visualizes the identified information in a compositional representation. *SEALANT* has been demonstrated to accurately identify inter-app vulnerabilities from hundreds of real-world Android apps and to effectively deliver the identified information to users. (Demo Video: https://youtu.be/E4lLQonOdUw)**

## I. Introduction

Modern mobile devices handle users' private information such as contacts, passwords, and text messages. Android, the most widely adopted mobile operating system today, implements various security measures to protect user information from attackers. However, Android still has known security vulnerabilities [1]. One well-known vulnerability resides in the design of Android's communication model [2], in which components within an app or across apps communicate by exchanging messages called *intents*. Specifically, if app developers do not carefully control the incoming and outgoing intents, inter-component communication (ICC) via intent can expose vulnerable surfaces to inter-app attacks, such as *intent spoofing* [2], *unauthorized intent receipt* [2], and *privilege escalation* [3].

A number of techniques have been proposed to detect inter-app vulnerabilities in Android [2], [4]–[9]. However, existing techniques target only single-app analysis [2], [4], [5] or support limited types of inter-app attacks [6]–[9]. While the state-of-the-art techniques [6], [7], [10] provide compositional analysis of multiple apps, they have been shown to encounter scalability problems in analyzing large numbers of apps, and/or to suffer from a potentially large number of false alarms. Furthermore, these techniques lack a systematic representation of their analysis results, which hinders effective and intuitive understanding of inter-app vulnerabilities by the end-users.

This paper presents *SEALANT*, a novel tool for automated analysis and visualization of Android inter-app vulnerabilities, which extends our prior work [11], [12]. *SEALANT* combines static analysis with visualization techniques to enable compositional security analysis and systematic assessment of a set of Android apps. Given a set of apps, *SEALANT* statically analyzes their architectural information and identifies vulnerable ICC paths between the apps by leveraging data-flow analysis and ICC pattern matching. It then visualizes the combination of architectural information of the target apps with the identified vulnerable ICC paths between them, enabling users to associate an appropriate action with each vulnerable path.

*SEALANT* is distinguished from the existing tools because (1) it simultaneously detects multiple types of inter-app attacks (i.e., *intent spoofing, unauthorized intent receipt,* and *privilege escalation*), (2) it extends the detection coverage and accuracy in identifying inter-app vulnerabilities, (3) it supports a compositional analysis that scales to hundreds of real-world apps, (4) it provides an efficient analysis by reusing prior analysis results, and (5) it effectively delivers to end-users a composition of large number of apps as well as their identified vulnerabilities via a systematic visualization.

The rest of this paper is organized as follows. Section II depicts examples of inter-app attacks that motivate our tool. Section III presents *SEALANT*'s architecture along with the description of major components and their implementations. Section IV highlights *SEALANT*'s key features and Section V demonstrates its evaluations. Section VI discusses related work and Section VII concludes the paper.

## II. Motivating Examples

In this section, we present two simplified examples of inter-app attacks: *intent spoofing* and *unauthorized intent receipt*.

Figure 1(a) depicts *intent spoofing*. Component `M1` of a malicious app `Malicious1` can send an intent to component `V2` of a victim app `Victim1` in order to exploit `Victim1`'s functionalities. For example, if `V2` is designed to dispatch an SMS to the number bundled in an incoming intent from `V1`, whenever `M1` injects a spoofed intent to `V2` along with unsafe number and text, `V2` will subsequently send an SMS to the unintended number. In this case, a vulnerable ICC path exists between `M1` and `V2`.

Figure 1(b) illustrates an *unauthorized intent receipt*. In case when an intent is sent or broadcast without particular

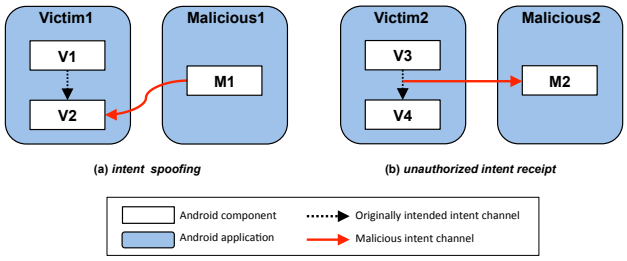


Fig. 1: Inter-App Attacks: (a) Intent Spoofing and (b) Unauthorized Intent Receipt

protection, any component can receive it by declaring attributes matching those of the intent. For example, component `V3` of `Victim2` is designed to broadcast an intent in order to deliver an account's information to the other component within the same app (i.e., `V4`). By listening to the intent, although not an intended receiver, component `M2` of a malicious app `Malicious2` can eavesdrop on the information bundled in the intent. In this case, a vulnerable ICC path exists between `V3` and `M2`.

As shown in the examples, inter-app attacks are not easy to distinguish from ordinary ICCs between apps. Since ICCs are essentially invisible to users, it is difficult for users to recognize where the attacks are actually launched. Furthermore, depending on the types of apps composing a user's device, vulnerable ICC paths will vary. These characteristics make detection and assessment of inter-app vulnerabilities quite challenging.

## III. SEALANT's Design and Implementation

*SEALANT* is a tool that automatically identifies ICC paths in a set of target apps that are vulnerable to inter-app attacks, and visualizes the identified paths using the architectural information of those target apps. *SEALANT* processes a set of APK[1] files as input and displays the output in a pre-defined representation.

As shown in Figure 2, *SEALANT* is designed to have two layers. (1) *SEALANT*'s *front-end* is responsible for providing the interactive user interface. The *SEALANT Client* component belongs to this layer. (2) *SEALANT*'s *back-end* is responsible for identifying vulnerable ICC paths from target apps, managing the extracted information, and transforming the information into a visually representable format. The back-end comprises three main components: *SEALANT Core*, *SEALANT Analyzer*, and *SEALANT Repository*. In the rest of this section, we will describe the details of *SEALANT*'s four components and their implementations.

### A. SEALANT Client

*SEALANT Client* facilitates a user's interaction with the *back-end* layer. Once the user selects a set of APK files that were automatically pulled from the user's device [16] or manually downloaded from online app stores, the *SEALANT Client* forwards the files to the *SEALANT Core* and informs the user of the analysis status (e.g., displaying the analysis progress).

The *SEALANT Client* renders the combined information obtained from the *SEALANT Core* (i.e., app models and identified vulnerable ICC paths) in tailorable visual notations. For example, a component may be depicted as a black dot and a vulnerable path as a red-line, as shown in Figure 3(a). The *SEALANT Client* provides interactive interfaces that enable users to customize display settings (e.g., size, font, and color) or highlight particular architectural elements (e.g., component or ICC path).

The *SEALANT Client* optionally contacts users to assess each vulnerable ICC path. Since *SEALANT*'s static analysis may over-approximate control-flow paths [11], which in turn may result in spurious vulnerable ICC paths in some cases, a user's assessment can reduce the falsely identified paths in subsequent analyses. Furthermore, referring to an expert user's assessments may aid non-expert users to isolate the falsely identified paths.

### B. SEALANT Core

*SEALANT Core* is responsible for controlling every operation and transforming the analyzed information into a visually representable format. It handles every request from *SEALANT Client*. If a set of APK files is requested, the *Core* component first checks if prior analysis results are stored in the *SEALANT Repository*. If the corresponding results exist, the *Core* reuses them. Otherwise, it decompiles the APK files and passes them to the *SEALANT Analyzer*. The *Core* subsequently combines and transforms the app models and vulnerable ICC paths it receives back from the *Analyzer* into a pre-defined format compatible with the *SEALANT Client*. The *Core* stores the user's assessments received from the *SEALANT Client* in the *SEALANT Repository*.

### C. SEALANT Analyzer

Given a set of apps, *SEALANT Analyzer* extracts each app's architectural information and summarizes the information as a model of each app. The *Analyzer* identifies vulnerable ICC paths between the apps using data-flow analysis and ICC pattern matching. It consists of two sub-components: *Model Extractor* and *Vulnerability Identifier*.

**Model Extractor** inspects each app's Android manifest file and bytecode to extract the app's architectural information (e.g., components, intent filters, permissions, and intents). For each app component, *Model Extractor* performs data-flow analysis between sensitive APIs [13] and ICC-call APIs [14]. A component that contains such a data-flow is marked as a *vulnerable component*. *Model Extractor* then builds a summary-based model that captures the information of each app, and stores every model in the *SEALANT Repository* for reuse in subsequent analyses.

**Vulnerability Identifier** analyzes vulnerable ICC paths from a set of app models. To this end, it first builds an ICC graph by matching intents and intent filters of each component based on the rules from Android's API reference documentation [15].

[1]APK (Android Package Kit) is an archive file format that distributes and installs Android apps on top of the Android operating system.

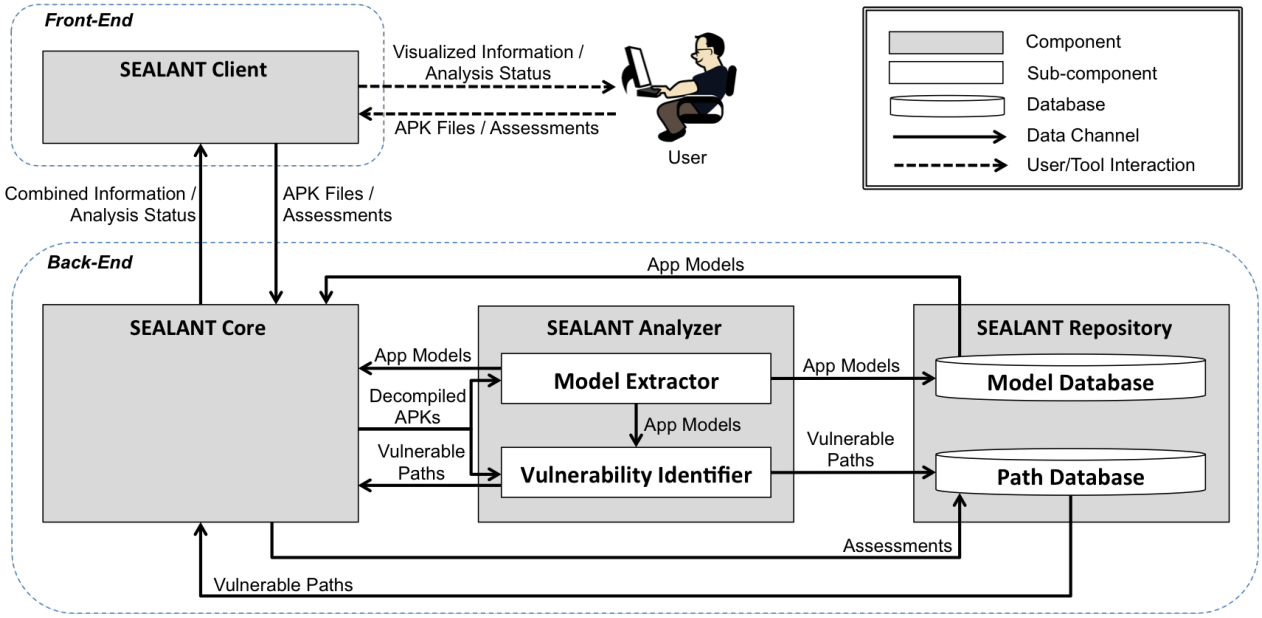


Fig. 2: *SEALANT*'s Architecture

In an ICC graph, a node represents a component and an edge is defined as a tuple $< s, r, i >$, where $s$ is a sender and $r$ a receiver component, and $i$ is an intent between them. *SEALANT* then traverses the ICC graph and marks an edge as vulnerable based on two criteria: (1) if there is a *vulnerable component* at one or both ends and (2) if it is a part of a particular compositional ICC patterns that may be vulnerable to inter-app attacks. Specifically, two different edges (one across apps and the other one within an app) that direct to the same component with the same type of intent represent a vulnerable ICC pattern susceptible to intent spoofing. Conversely, two different edges (one across apps and the other one within an app) that start from the same component with the same type of intent represent a vulnerable ICC pattern susceptible to unauthorized intent receipt. The identified vulnerable ICC paths are stored in the *SEALANT Repository* for reuse in subsequent analyses.

### D. SEALANT Repository

*SEALANT Repository* maintains the extracted app models in its *Model Database* sub-component and the identified vulnerable ICC paths in the *Path Database* sub-component.

**Model Database** manages each app's model by its package name and versions in order to enable reusing the extracted information for further analyses. When apps are installed or updated on a user's device subsequent to running *SEALANT*, *SEALANT* extracts the architectural information only from the installed or updated apps, and reuses the prior analysis results for the rest. This can greatly reduce the total analysis time (as evaluated in Section V).

**Path Database** maintains the identified vulnerable ICC paths. For each vulnerable ICC path, it also maintains each user's assessment that indicates whether the ICC path is indeed exposed to inter-app attacks or not.

### E. Implementation

We have implemented the *SEALANT Analyzer* as a stand-alone Java application that uses a set of APK files as its input. It combines approximately 3,000 newly written LOC with off-the-shelf tools. Specifically, to enable a complete extraction of architectural information from apps, the *Analyzer* integrates two static analysis tools: IC3 [17] and COVERT [10]. To perform data-flow analysis on each component, the *Analyzer* leverages FlowDroid [18], an intra-component taint analysis tool for Android. The *Analyzer* combines the outputs from those tools into complete app models. The app models for architectural objects and identified vulnerable ICC paths are, respectively, transformed into two different JSON files with pre-defined formats. The JSON file for vulnerable ICC paths is also compatible with *Interceptor* [11], an extension module we have developed for the Android framework to control runtime ICCs based on a given list of ICC paths.

We implemented the *SEALANT Core* using Java and shell script code that integrates different components, totaling around 3,600 LOC.

We implemented the *SEALANT Repository* as a file system-based repository. It maintains three different types of files: (1) app models, (2) vulnerable ICC paths, and (3) assessments of each vulnerable ICC path.

Finally, we implemented the *SEALANT Client* as a web application, using HTML5 with the off-the-shelf JavaScript libraries that include jQuery [19] for event handling web pages, Bootstrap [20] for developing a responsive web application, and D3 [21] for a data-driven visualization. The size of the *SEALANT Client* is approximately 4,500 LOC. We extended the display notation for the *SEALANT Client* by specifying a meta-model in D3. The notation includes the essential elements representing Android app architectures (i.e., components, apps, and ICC paths).

## IV. Key Tool Features

*SEALANT*'s key features are intended to automatically identify ICC vulnerabilities among a given set of apps and effectively deliver the identified information to users. The features also enable users to assess each identified vulnerable ICC path, selectively focus on particular architectural elements (e.g., paths, components, or apps), and reuse prior analysis results. The features that can be accessed via the *SEALANT Client* are shown in Figure 3.

**Automated Detection of Vulnerabilities.** Given a set of APK files or app models from prior analyses, *SEALANT* automatically processes the detection of inter-app vulnerabilities. The entire detection process takes place in the back-end. During the process, progress bars display the status of each analysis step (i.e., extraction of architectural information and identification of vulnerable ICC paths). Once the analysis is completed, *SEALANT* provides options for exporting the identified vulnerabilities in a textual format or visualizing them in pre-defined notations.

**Visualization of Architecture and Vulnerabilities.** To effectively visualize a number of components in Android apps, *SEALANT* displays the integrated and compositional view of target apps as shown in Figure 3(a). *SEALANT* depicts each component as a labeled dot, each app as a cluster of components, and each ICC path between a pair of components as a *gray* line connecting the respective dots. The vulnerable ICC paths are presented as colored lines, with different colors for each vulnerability type: intent spoofing is *red*, unauthorized intent receipt is *blue*, and privilege escalation is *green*. A user can customize display options such as font size, display size, and line tension. As part of the visualization feature, if a user hovers the pointer over an app or a component name, *SEALANT* shows its detailed information, such as type, intent filters, and permissions. As shown in Figure 3(b), *SEALANT* also presents detailed information for each vulnerable ICC path, and a user can assess whether a given path is indeed unsafe or not. The user assessments can provide a reference point for future analyses, as detailed below.

**Selective Visualization.** For a set of apps that contain a large number of components and ICC paths, displaying all the components and paths may be incomprehensible to users. To handle this, *SEALANT* provides a selective visualization that allows a user to focus on a particular set of apps, components, or vulnerabilities. For example, if a user clicks on a desired component's name, *SEALANT* highlights the ICC paths (and the corresponding components) that are connected to it, and fades out the other parts. *SEALANT* also enables users to choose different visualization types that use different notations and topologies.

**Reuse of Prior Analyses.** When apps are installed or updated, rerunning the *SEALANT Analyzer* is required to identify vulnerable paths in the new ICC graph. To effectively handle this, *SEALANT* supports reusing prior analysis results by maintaining the models for apps and vulnerable ICC paths, respectively. Whenever the analysis is requested, *SEALANT* automatically checks if the same apps' models or corresponding vulnerable ICC path information exist in its *Repository*. Consequently, the analysis time will primarily depend on the number of newly analyzed apps.

## V. Evaluation

In this section, we evaluate *SEALANT*'s accuracy, scalability, performance, and usability.

**Accuracy.** Our prior research [11] has evaluated accuracy of *SEALANT* in terms of detecting vulnerable ICC paths from real-world apps. Specifically, we created a test suite comprising the apps from existing benchmarks and externally developed apps [7], [22] (in total, 135 apps with 59 vulnerable ICC paths). From the test suite, *SEALANT* detected vulnerable ICC paths with 100% precision[2] and 95% recall[3], both of which are higher than those of existing tools (SEPAR [7]: 50% and 8%, respectively; IccTA [6]: 100% and 8%, respectively).

**Scalability.** To evaluate *SEALANT*'s scalability, we extended the above test suite by randomly including apps from public sources [23], [24]. The selected 1,150 apps were divided into 23 non-overlapping bundles comprising 50 apps each. These bundles are larger than the number of apps an average Android user is shown to regularly use each month [25]. From the bundles, *SEALANT* flagged 86 ICC paths with 93% precision. The results demonstrated that *SEALANT* successfully works on representative sets of real-world apps with improved scalability as compared to existing tools.

**Performance.** We evaluated *SEALANT*'s performance in detecting vulnerable ICC paths [11]. From the extended test suite used above, we randomly selected four categories of ten different bundles of apps. A category comprised 25, 50, 75, or 100 apps. As shown in Figure 4, since *SEALANT* maintains a summary-based model of each app, as expected the analysis time scaled linearly with the number of apps and components. On average, extracting architectural information from each app took 77.95$s$ and identifying vulnerable ICC paths took 1.08$s$ per app. Although the extraction is relatively time-consuming, *SEALANT*'s feature for reusing prior analysis can reduce up to 98.63% of the analysis time in subsequent analyses.

**Usability.** To confirm that *SEALANT*'s visualization helps the users' understanding of inter-app vulnerabilities, we conducted a user study. Our participants included 30 computer science graduate students at the University of Southern California, recruited via an e-mail list. The background survey indicated that 25 of the participants (83.33%) used Android as their primary mobile platform while the remaining 5 (16.67%) had prior experience using it. The analysis results of vulnerable ICC paths (of ten Android apps that form five inter-app vulnerabilities) were given to each participant via two different representation methods: a textual list and *SEALANT*'s visualization. Participants were asked to evaluate the level of difficulty in understanding vulnerabilities for each method, using a 7-point Likert scale (1 = very difficult, 7 = very easy). We also asked

[2]Precision: the ratio of vulnerable ICC paths to identified ICC paths.
[3]Recall: the ratio of identified to all vulnerable ICC paths.

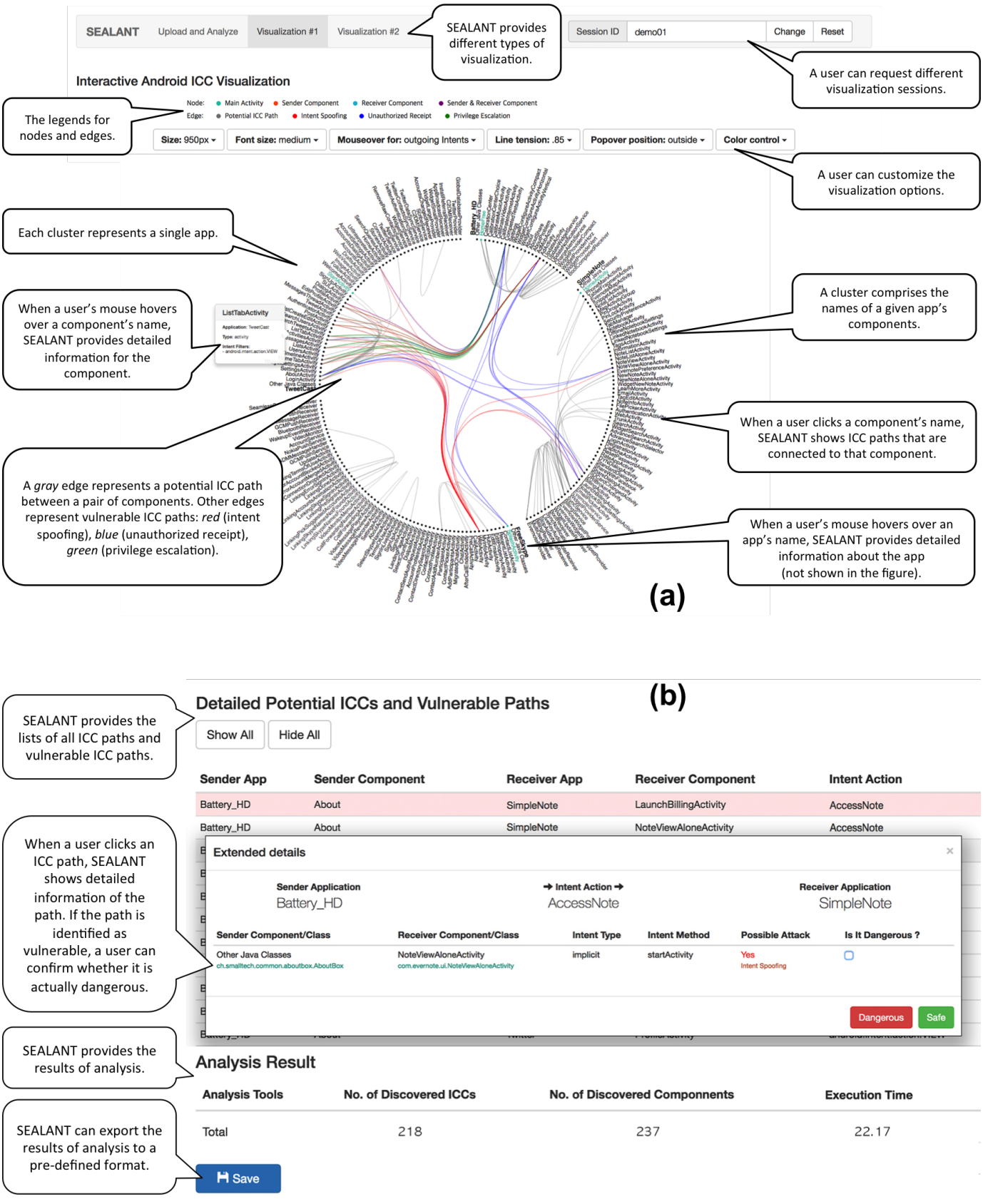


Fig. 3: Two Different Screenshots of *SEALANT*'s User Interface

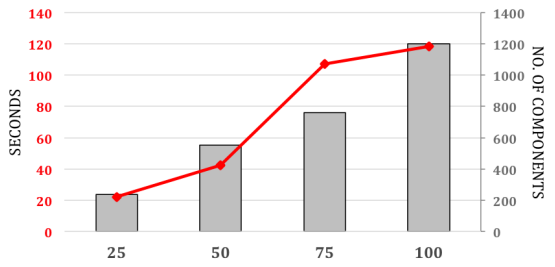


Fig. 4: *SEALANT*'s Performance on Different Numbers of Apps and Components

them about the factors that affected their evaluation. According to the responses, the mean degrees of *SEALANT*'s visualization (Mean = 5.43, Std. = 1.66) was higher than that of textual representation (Mean = 2.82, Std. = 1.60). The factors that affect their preference for *SEALANT*'s visualization were *intuitiveness* (54.55%), *ease of understanding* (31.82%), *compositional view* (9.09%), and others (4.54%). The results support the conclusion that *SEALANT*'s visualization successfully aids a user's understanding of inter-app vulnerabilities.

## VI. Related Work

**ICC analysis** is used to detect Android's inter-app vulnerabilities. Although AmanDroid [9] identifies privacy leaks by tracking interactions between components, it has shown its inaccuracy when it comes to *Content Provider* components and certain ICC methods. IccTA [6] detects privacy leaks between components using a taint-flow analysis. While its bytecode instrumentation resolves the ICC paths between components, it is not scalable to a large number of apps. COVERT [10] introduces a compositional analysis of a set of apps to identify permission leakage. However, it does not target other types of inter-app attacks, such as unauthorized intent receipt.

**ICC visualization** is employed to aid understanding the systems that use ICC mechanism. ViVA [12] is our prior visualization and analysis tool for event-based systems, which visualizes a target system's architectural information, message-based dependencies between components, and runtime message exchanges. However, it neither detects inter-app vulnerabilities nor supports Android. Although COVERT has a feature that visualizes its vulnerability analysis results, it does not support a compositional visualization of a large number of apps or a dynamic user interface.

## VII. Conclusion

Our experimental results of applying *SEALANT* to hundreds of real-world Android apps have demonstrated that it accurately identifies vulnerable ICC paths. Our user study also has shown that it successfully aids users in understanding the identified vulnerabilities. We plan to extend *SEALANT* to support other types of inter-app attacks that exploit covert channels in the Android core system components, such as the file system. Another potential direction for future work is evaluating different types of displays (e.g., different notations and colors) to find the most efficient way of delivering Android inter-app vulnerabilities.

## References

[1] "2012 Norton Cybercrime Report," Symantec Corporation, September 2012. [Online]. Available: http://www.norton.com/2012cybercrimereport
[2] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2011.
[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks," Technische Universität Darmstadt, Tech. Rep. TR-2011-04.
[4] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proceedings of the USENIX Conference on Security*, 2013.
[5] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
[6] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android App," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.
[7] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android," in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.
[8] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android Taint Flow Analysis for App Sets," in *Proceedings of the 3rd International Workshop on the State of the Art in Java Program Analysis*, 2014.
[9] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.
[10] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: Compositional Analysis of Android Inter-App Permission Leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.
[11] Y. K. Lee, J. y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A SEALANT for Inter-app Security Holes in Android," in *Proceedings of the 39th International Conference on Software Engineering*, 2017.
[12] Y. K. Lee, J. y. Bang, J. Garcia, and N. Medvidovic, "ViVA: A Visualization and Analysis Tool for Distributed Event-based Systems," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.
[13] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks," in *Proceedings of the Network and Distributed System Security Symposium*, 2014.
[14] "android.app | Android Developers." [Online]. Available: http://developer.android.com/reference/android/app/package-summary.html
[15] "Intents and Intent Filters | Android Developers." [Online]. Available: https://developer.android.com/guide/components/intents-filters.html
[16] "Apk Extractor." [Online]. Available: https://play.google.com/store/apps/details?id=com.ext.ui&hl=en/
[17] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.
[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, 2014.
[19] "jQuery." [Online]. Available: https://jquery.com/
[20] "Bootstrap." [Online]. Available: http://getbootstrap.com/
[21] "D3.js - Data-Driven Documents." [Online]. Available: https://d3js.org/
[22] "DroidBench." [Online]. Available: https://github.com/secure-software-engineering/DroidBench
[23] "Google Play." [Online]. Available: http://play.google.com/store/apps
[24] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.
[25] "So Many Apps, So Much Time For Entertainment." [Online]. Available: http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html