Towards Automatic Generation of Security-Centric Descriptions for Android Apps

Mu Zhang[$*] Yue Duan[†] Qian Feng[†] Heng Yin[†]

[$]NEC Labs America, Inc. [†]Department of EECS, Syracuse University, USA
[$]mu@nec-labs.com [†]{yuduan,qifeng,heyin}@syr.edu

ABSTRACT

To improve the security awareness of end users, Android markets directly present two classes of literal app information: 1) permission requests and 2) textual descriptions. Unfortunately, neither can serve the needs. A permission list is not only hard to understand but also inadequate; textual descriptions provided by developers are not security-centric and are significantly deviated from the permissions. To fill in this gap, we propose a novel technique to automatically generate security-centric app descriptions, based on program analysis. We implement a prototype system, DESCRIBEME, and evaluate our system using both DroidBench and real-world Android apps. Experimental results demonstrate that DESCRIBEME enables a promising technique which bridges the gap between descriptions and permissions. A further user study shows that automatically produced descriptions are not only readable but also effectively help users avoid malware and privacy-breaching apps.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Android; Malware prevention; Textual description; Program analysis; Subgraph mining; Natural language generation

1. INTRODUCTION

As usage of Android platform has grown, security concerns have also increased. Malware [12, 43, 45], software vulnerabilities [17, 20, 24, 44] and privacy issues [14, 46] severely violate end user security and privacy.

Unlike traditional desktop systems, Android provides end users with an opportunity to proactively accept or deny the installation of any app to the system. As a result, it is essential that the users become aware of app behaviors so as to make appropriate decisions. To this end, Android markets directly present the consumers with two classes of information regarding each app: 1) the requested Android permissions and 2) textual description provided by the app's developer. Unfortunately, neither can fully serve this need.

Permission requests are not easy to understand. First, prior study [15] has shown that few users are cautious or knowledgeable enough to comprehend the security implications of Android permissions. Second, a permission list merely tells the users which permissions are used, but does not explain *h*ow they are used. Without such knowledge, one cannot properly assess the risk of allowing a permission request. For instance, both a benign navigation app and a spyware instance of the same app can require the same permission to access GPS location, yet use it for completely different purposes. While the benign app delivers GPS data to a legitimate map server upon the user's approval, the spyware instance can periodically and stealthily leak the user's location information to an attacker's site. Due to the lack of context clues, a user is not able to perceive such differences via the simple permission enumeration.

Textual descriptions provided by developers are not security-centric. There exists very little incentive for app developers to describe their products from a security perspective, and it is still a difficult task for average developers (usually inexperienced) to write dependable descriptions. Malware authors can also intentionally hide malice from innocent users by providing misleading descriptions. Studies [26, 28] have revealed that the existing descriptions deviate considerably from requested permissions. Thus, developer-driven description generation cannot be considered trustworthy.

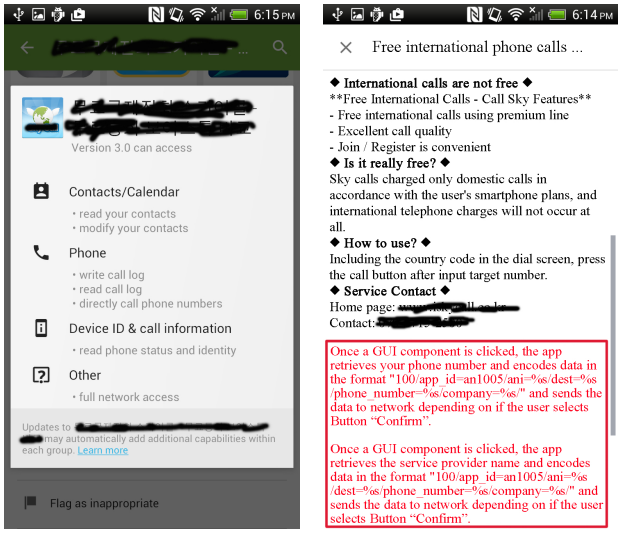
To address this issue, we propose a novel technique to automatically generate app descriptions which accurately describe the security-related behaviors of Android apps. To interpret panoramic app behaviors, we extract *security behavior graphs* as high-level program semantics. To create concise descriptions, we further condense the graphs by mining and compressing the frequent subgraphs. As we traverse and parse these graphs, we leverage *Natural Language Generation* (NLG) to automatically produce concise, human-understandable descriptions.

A series of efforts have been made to describe the functionalities of traditional Java programs as human readable text via NLG. Textual summaries are automatically produced for methods [30], method parameters [32], classes [25], conditional code snippets [11] and algorithmic code structures [31] through program analysis and comprehension. However, these studies focus upon depicting the intra-procedural structure-based operations. In contrast, our technique presents the whole-program's semantic-level activities. Fur-

*This work was conducted while Mu Zhang was a PhD student at Syracuse University, advised by Prof. Heng Yin.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813669.



(a) **Permission Requests.** (b) **Old+New Descriptions.**

Figure 1: **Metadata of the Example App.**

thermore, we take the first step towards automating Android app description generation for security purposes.

We implement a prototype system, DESCRIBEME, in 25 thousand lines of Java code. Our behavior graph generation is built on top of Soot [8], while our description production leverages an NLG engine [7] to realize texts from the graphs. We evaluate our system using both DroidBench [3] and real-world Android apps. Experimental results demonstrate that DESCRIBEME is able to effectively bridge the gap between descriptions and permissions. A further user study shows that our automatically-produced descriptions are both readable and effective at helping users avoid malware and privacy-breaching apps.

Natural language generation is in general a hard problem, and it is an even more challenging task to describe app behaviors to average users in a comprehensive yet concise, and most importantly, human-readable manner. While we have demonstrated promising results, we do not claim that our system is fully mature and has addressed all the challenges. However, we believe that we have made a solid step towards this goal. We also hope the report of our experience can attract more attention and stimulate further research.

In summary, this paper makes the following contributions:

- We propose a novel technique that automatically describes security-related app behaviors to the end users in natural language. To the best of our knowledge, we are the first to produce Android app descriptions for security purpose.
- We implement a prototype system, DESCRIBEME, that combines multiple techniques, including program analysis, subgraph mining and natural language generation, and adapts them to the new problem domain, which is to systematically create expressive, concise and human-readable descriptions.
- Evaluation and user study demonstrate that DESCRIBEME significantly improves the expressiveness of textual descriptions, with respect to security-related behaviors.

2. OVERVIEW

2.1 Problem Statement

Figure 1a and Figure 1b demonstrate the two classes of descriptive metadata that are associated with an Android app available via Google Play. The app shown leaks the user's phone number and

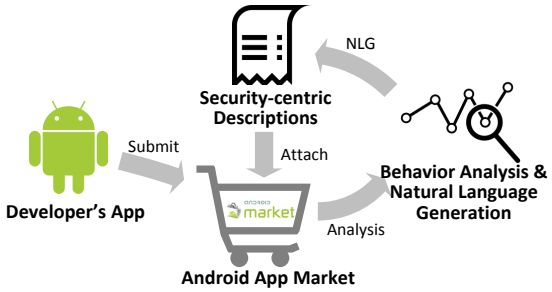


Figure 2: Deployment of DESCRIBEME

service provider to a remote site. Unfortunately, neither of these two pieces of metadata can effectively inform end users of the risk.

The permission list (Figure 1a) simply enumerates all of the permissions requested by the app while replacing permission primitives with straightforward explanations. Besides, it can merely tell users that the app uses two separate permissions, `READ_PHONE_STATE` and `INTERNET`, but cannot indicate that these two permissions are used consecutively to send out phone number. The textual descriptions are not focused on security. As depicted in the example (the top part in Figure 1b), developers are more interested in describing the app's functionalities, unique features, special offers, use of contact information, etc. Prior studies [26,28] have revealed significant inconsistencies between app descriptions and permissions.

We propose a new technique, DESCRIBEME, which addresses these shortcomings and can automatically produce complementary security-centric descriptions for apps in Android markets. It is worth noting that we do not expect to replace the developers' descriptions with ours. Instead, we hope to provide additional app information that is written from a security perspective. For example, as shown in the bottom part of Figure 1b, our security-sensitive descriptions are attached to the existing ones. The new description states that the app retrieves the phone number and writes data to network, and therefore indicates the privacy-breaching behavior. Notice that Figure 1b only shows a portion of our descriptions, and a complete version is depicted in Appendix A.

We expect to primarily deploy DESCRIBEME directly into the Android markets, as illustrated in Figure 2. Upon receiving an app submission from a developer, the market drives our system to analyze the app and create a security-centric description. The generated descriptions are then attached to the corresponding apps in the markets. Thus, the new descriptions, along with the original ones, are displayed to consumers once the app is ready for purchase.

Given an app, DESCRIBEME aims at generating natural language descriptions based on security-centric program analyses. More specifically, we achieve the following design goals:

- **Semantic-level Description.** Our approach produces descriptions for Android apps solely based upon their program semantics. It does not rely upon developers' statements, users' review, or permission listings.
- **Security-centric Description.** The generated descriptions focus on the security and privacy aspects of Android apps. They do not exhaustively describe all program behaviors.
- **Human Readability.** The crafted descriptions are natural language based scripts that are comprehensible to end users. Besides, the descriptive texts are concise. They do not contain superfluous components or repetitive elements.

2.2 Architecture Overview

Figure 3 depicts the workflow of our automated description generation. This takes the following steps:

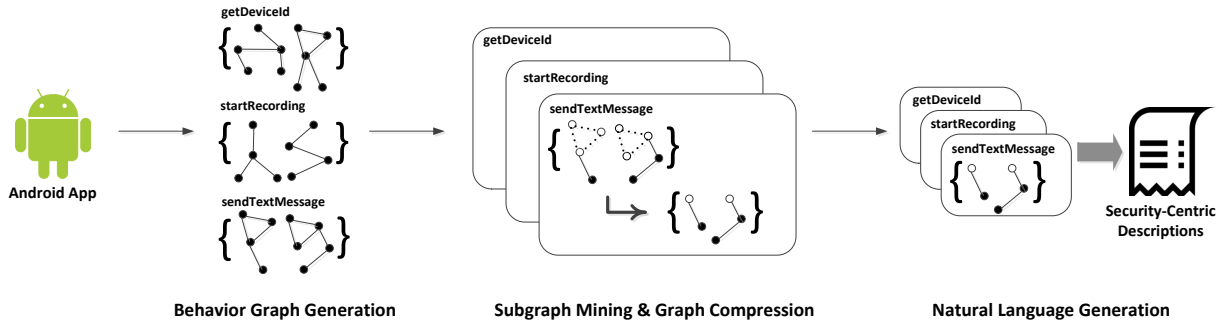


Figure 3: Overview of DESCRIBEME

(1) **Behavior Graph Generation.** Our natural language descriptions are generated via directly interpreting program behavior graphs. To this end, we first perform static program analyses to extract behavior graphs from Android bytecode programs. Our program analyses enable a condition analysis to reveal the triggering conditions of critical operations, provide entry point discovery to better understand the API calling contexts, and leverage both forward and backward dataflow analyses to explore API dependencies and uncover constant parameters. The result of these analyses is expressed via *Security Behavior Graphs* that expose security-related behaviors of Android apps.

(2) **Subgraph Mining & Graph Compression.** Due to the complexity of object-oriented, event-driven Android programs, static program analyses may yield sizable behavior graphs which are extremely challenging for automated interpretation. To address this problem, we next reduce the graph size using subgraph mining. More concretely, we first leverage data mining based technique to discover the *frequent subgraphs* that bear specific behavior patterns. Then, we compress the original graphs by substituting the identified subgraphs with single nodes.

(3) **Natural Language Generation.** Finally, we utilize *natural language generation* technique to automatically convert the semantically rich graphs to human understandable scripts. Given a compressed behavior graph, we traverse all of its paths and translate each graph node into a corresponding natural language sentence. To avoid redundancy, we perform sentence aggregation to organically combine the produced texts of the same path, and further assemble only the distinctive descriptions among all the paths. Hence, we generate descriptive scripts for every individual behavior graph derived from an app and eventually develop the full description for the app.

3. SECURITY BEHAVIOR GRAPH

3.1 Security-related Behavioral Factors

We consider the following four factors as essential when describing the security-centric behaviors of an Android app sample:

1) **API call and Dependencies.** Permission-related API calls directly reflect the security-related app behaviors. Besides, the dependencies between certain APIs indicate specific activities.

2) **Condition.** The triggering conditions of certain API calls imply potential security risks. The malice of an API call is sometimes dependent on the absence or presence of specific preconditions. For instance, a missing check for user consent may indicate unwanted operations; a condition check for time or geolocation may correspond to trigger-based malware.

3) **Entry point.** Prior studies [12, 40] have demonstrated that the entry point of a subsequent API call is an important security indicator. Depending on the fact an entry point is a user interface or background event handler, one can infer whether the user is aware that such an API call has been made or not.

4) **Constant.** Constant parameters of certain API calls are also essential to security analysis. The presence of a constant argument or particular constant values should arouse analysts' suspicions.

3.2 Formal Definition

To consider all these factors, we describe app behaviors using *Security Behavior Graphs* (`SBG`). An `SBG` consists of behavioral operations where some operations have data dependencies.

Definition 1. A *Security Behavior Graph* is a directed graph $G = (V, E, \alpha)$ over a set of operations Σ, where:

• The set of vertices V corresponds to the behavioral operations (i.e., APIs or behavior patterns) in Σ;

• The set of edges $E \subseteq V \times V$ corresponds to the *data dependencies* between operations;

• The labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the labels of corresponding semantic-level operations, where each label is comprised of 4 elements: behavior name, entry point, constant parameter set and precondition list.

Notice that a behavior name can be either an API prototype or a behavior pattern ID. However, when we build `SBG`s using static program analysis, we only extract API-level dependency graphs (i.e., the raw `SBG`s). Then, we perform frequent subgraph mining to identify common behavior patterns and replace the subgraphs with pattern nodes. This will be further discussed in Section 4.

3.3 `SBG` of Motivating Example

Figure 4 presents an `SBG` of the motivating example. It shows that the app first obtains the user's phone number (`getLine1Number()`) and service provider name (`getSimOperatorName()`), then encodes the data into a format string (`format(String,byte[])`), and finally sends the data to network (`write(byte[])`).

All APIs here are called after the user has clicked a GUI component, so they share the same entry point, `OnClickListener.onClick`. This indicates that these APIs are triggered by user.

The sensitive APIs, including `getLine1Number()`, `getSimOperatorName()` and `write(byte[])`, are predominated by a UI-related condition. It checks whether the clicked component is a `Button` object of a specific name. There exist two security implications behind this information: 1) the app is usually safe to use, without leaking the user's phone number; 2) a user should be cautious when she is about to click this specific button, because the subsequent actions can directly cause privacy leakage.

The encoding operation, `format(String,byte[])`, takes a constant format string as the parameter. Such a string will later be used

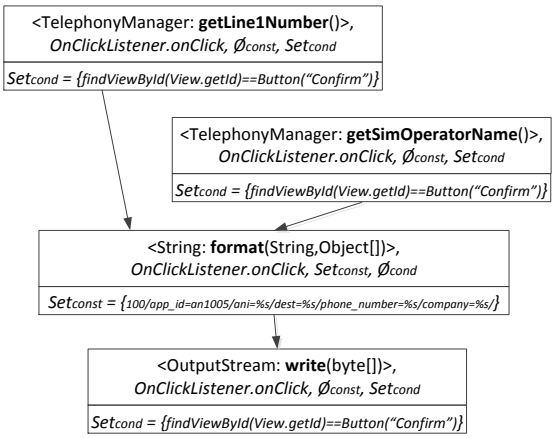


Figure 4: An example SBG

to compose the target URL, so it is an important clue to understand the scenario in which the privacy-related data is used.

3.4 Graph Generation

To generate an SBG, we have implemented a static analysis tool, built on top of Soot [8], in 22K lines of code.

To extract API data dependencies and constant parameters, we perform context-sensitive, flow-sensitive, and interprocedural dataflow analysis. In theory, we take the same approach as the prior works [9,24,34,40]. Our analysis first considers the dataflow within individual program "splits" and then conducts inter-split analysis with respect to Android Activity/Service lifecycles. Notice that our analysis does not support implicit dataflow at this point.

We use the algorithm in prior work [40] to discover entry points. We perform callgraph analysis while taking asynchronous calls into consideration. Thus, the identified entry points can faithfully reflect whether an API is triggered by a user action.

Condition Reconstruction. We then perform both control-flow and dataflow analyses to uncover the triggering conditions of sensitive APIs. All conditions, in general, play an essential role in security analysis. However, we are only interested in certain trigger conditions for our work. This is because our goal is to generate human understandable descriptions for end users. This implies that an end user should be able to naturally evaluate the produced descriptions, including any condition information. Hence, it is pointless if we generate a condition that cannot be directly observed by a user.

Consequently, our analysis is only focused on three major types of conditions that users can directly observe. 1) *User Interface.* An end user actively communicates with the user interface of an app, and therefore she directly notices the UI-related conditions, such as a click on a specific button. 2) *Device status.* Similarly, a user can also notice the current phone status, such as WIFI on/off, screen locked/unlocked, speakerphone on/off, etc. 3) *Natural environment.* A user is aware of environmental factors that can impact the device's behavior, including the current time and geolocation.

The algorithm for condition extraction is presented in Algorithm 1. This algorithm accepts a supergraph SG as the input and produces $Set_{<a,c>}$ as the output. SG is derived from callgraph and control-flow analyses; $Set_{<a,c>}$ is a set of $< a, c >$ pairs, each of which is a mapping between a sensitive API and its conditions.

Given the supergraph SG, our algorithm first identifies all the sensitive API statements, Set_{api}, on the graph. Then, it discovers the conditional predecessors Set_{pred} (e.g., IF statement) for each API statement via GetConditionalPredecessors(). Conditional predecessor means that it is a predominator of that API statement but the API statement is not its postdominator. Intuitively, it means the occurrence of that API statement is indeed conditional and depends on the predicate within that predecessor. Next, for every conditional statement $pred$ in Set_{pred}, it performs backward dataflow analysis on all the variables defined or used in its predicate. The result of BackwardDataflowAnalysis() is a data dependency graph DDG, which represents the dataflow from the variable definitions to the conditional statement. The algorithm further calls ExtractCondition(), which traverses this DDG and extracts the conditions Set_{cond} for the corresponding api statement. In the end, the API/conditions pair $< api, Set_{cond} >$ is merged to output set $Set_{<a,c>}$.

Algorithm 1 Condition Extraction for Sensitive APIs

$SG \leftarrow$ Supergraph
$Set_{<a,c>} \leftarrow$ null
$Set_{api} \leftarrow$ {sensitive API statements in the SG}
for $api \in Set_{api}$ **do**
  $Set_{pred} \leftarrow$ GetConditionalPredecessors(SG,api)
  for $pred \in Set_{pred}$ **do**
    for $\forall var$ defined and used in $pred$ **do**
      $DDG \leftarrow$ BackwardDataflowAnalysis(var)
      $Set_{cond} \leftarrow$ ExtractCondition(DDG, var)
      $Set_{<a,c>} \leftarrow Set_{<a,c>} \cup \{< api, Set_{cond} >\}$
    end for
  end for
end for
output $Set_{<a,c>}$ as a set of $< API, conditions >$ pairs

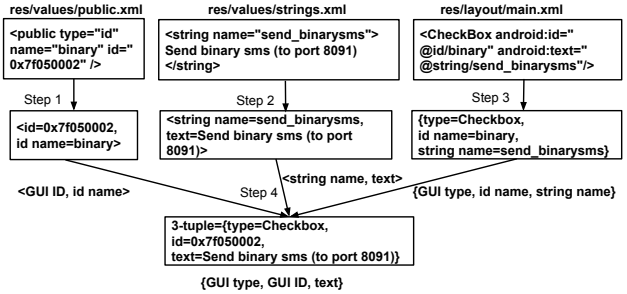


Figure 5: Extraction of UI Information from Resource Files

We reiterate that ExtractCondition() only focuses on three types of conditions: user interface, device status and natural environment. It determines the condition types by examining the API calls that occur in the DDG. For instance, an API call to findViewById() indicates the condition is associated with GUI components. The APIs retrieving phone states (e.g., isWifiEnabled(), isSpeakerphoneOn()) are clues to identify phone status related conditions. Similarly, if the DDG involves time- or location-related APIs (e.g., getHours(), getLatitude()), the condition is corresponding to natural environment.

User Interface Analysis in Android Apps. We take special considerations when extracting UI-related conditions. Once we discover such a condition, we expect to know exactly which GUI component it corresponds to and what text is actually displayed to users.

In order to retrieve GUI information, we perform an analysis on the Android resource files for the app. Our UI resource analysis is different from the prior work (i.e., AsDroid [21]) in that AsDroid examines solely the GUI-related call chains while we aim for the depiction of application-wide behaviors. Therefore, AsDroid only needs to correlate GUI texts to program entry points and then de-

tect any conflicts on the callgraph. In contrast, we have to further associate the textual resources to specific conditional statements, so that we can give concrete meaning to the subsequent program logics preceded by the conditions. Besides, the previous work did not consider those GUI callbacks that are registered in XML layout files, whereas we handle both programmatically and statically registered callbacks in order to guarantee the completeness.

Figure 5 illustrates how we perform UI analysis. This analysis takes four steps. First, we analyze the `res/values/public.xml` file to retrieve the mapping between the GUI ID and GUI name. Then, we examine the `res/values/strings.xml` file to extract the string names and corresponding string values. Next, we recursively check all layout files in the `res/layout/` directory to fetch the mapping of GUI type, GUI name and string name. At last, all the information is combined to generate a set of 3-tuples {GUI type, GUI ID, string value}, which is queried by `ExtractCondition()` to resolve UI-related conditions.

Notice that dynamically generated user interfaces are not handled through our static analysis. To address this problem, more advanced dynamic analysis is required. We leave this for future study.

Condition Solving. Intuitively, we could use a constraint solver to compute predicates and extract concrete conditions. However, we argue that this technique is not suitable for our problem. Despite its accuracy, a constraint solver may sometimes generate excessively sophisticated predicates. It is therefore extremely hard to describe such complex conditions in a human readable manner. Thus, we instead focus on simple conditions, such as equations or negations, because their semantics can be easily expressed in natural language.

Therefore, once we have extracted the definitions of condition variables, we further analyze the equation and negation operations to compute the condition predicates. To this end, we analyze how the variables are evaluated in conditional statements. Assume such a statement is `if(hour == 8)`. In its predicate `(hour == 8)`, we record the constant value 8 and search backwardly for the definition of variable `hour`. If the value of `hour` is received directly from API call `getHours()`, we know that the condition is `current time is equal to 8:00am`. For conditions that contain negation, such as a condition like `WIFI is NOT enabled`, we examine the comparison operation and comparison value in the predicate to retrieve the potential negation information. We also trace back across the entire def-use chain of the condition variables. If there exists a negation operation, we negate the extracted condition.

One concern for our condition extraction is that attackers with prior knowledge of our system can deliberately create complex predicates to disable the analysis. However, we argue that even if the logics cannot be resolved, the real malicious API calls will still be captured and described alongside with other context and dependency information.

4. SUBGRAPH MINING & COMPRESSION

Static analysis sometimes results in huge behavior graphs. To address this problem, we identify higher-level behavior patterns from raw `SBG`s so as to compress them and produce concise descriptions.

4.1 Frequent Behavior Mining

Experience tells us certain APIs are typically used together to achieve particular functions. For example, `SMSManager.getDefault()` always happens before `SMSManager.sendTextMessage()`. We expect to extract these behavior patterns, so that we can describe each pattern as an entirety instead of depicting every API included. To this end, we first discover the common subgraph patterns, and later compress the original raw graphs by collapsing pattern nodes.

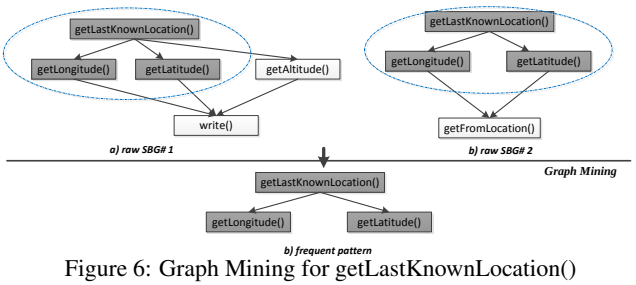


Figure 6: Graph Mining for getLastKnownLocation()

We leverage the graph mining technique to extract the frequent behavior patterns in raw `SBG`s. Given the raw `SBG` dataset $S = \{G_1, G_2, \ldots, G_N\}$, where $N = |S|$ is the size of the set, we hope to discover the frequent subgraphs appearing in S. To quantify the subgraph frequency, we introduce the support value $support_g$ for a subgraph g. Suppose the set of graphs, containing subgraph g, is defined as $S_g = \{G_i | g \subseteq G_i, 1 \leq i \leq N\}$. Then, $support_g = |S_g|/N$, where $|S_g|$ denotes the cardinality of S_g. It demonstrates the proportion of graphs in S that contains the subgraph g. Consequently, we define the frequent subgraphs appearing in S as:

$$\mathcal{F}(S, \rho) = \{g \mid support_g \geq \rho\} \tag{1}$$

, where ρ is a threshold. Therefore, to discover a frequent behavior pattern is to select a ρ and find all subgraphs whose $support_g \geq \rho$.

A naive way to solve this problem is to directly apply behavior mining to the entire behavior graph set S, and extract the frequent behaviors shared by all the graphs. However, there exist two problems in this solution. First, a behavior graph includes too many attributes in a node. As a result, we cannot really learn the common patterns when considering every attribute. In fact, we are more interested in the correlations of API calls, and thus can focus only on their topological relations. Second, critical behaviors may not be discovered as patterns because they do not frequently happen over all raw `SBG`s. To uncover those critical yet uncommon API patterns, we conduct an "API-oriented" mining and extract the frequent patterns that are specific to individual APIs.

Given an API θ, the "API-oriented" behavior mining operates on the subset $S/\theta = \{G_1, G_2, \ldots, G_M\}$, where $G_1, G_2, \ldots, G_M$ are raw `SBG`s in S containing the API θ. Hence, we need to select an individual support threshold ρ_θ for each S/θ. The quality of discovered patterns is then determined by these thresholds.

To achieve a better quality, we need to consider two factors: support value and graph size. On the one hand, we hope a discovered pattern is prevalent over apps and therefore bears a higher support value. On the other hand, we also expect an identified subgraph is large enough to represent meaningful semantic information. To strike a balance between these two factors, we utilize the data compression ratio [27] to quantify the subgraph quality. Given an API θ, g is any subgraph that contains θ; S_g is the set of graphs that contain subgraph g; and $G_{\bar{g}}$ is the compressed graph of G, where subgraph g has been replaced. Then, our goal is to optimize the total compression ration (TCR) by adjusting the threshold ρ_θ:

$$\begin{aligned} \textbf{max} \quad & TCR(\theta, \rho_\theta) = \sum_{G,g} (1 - |G_{\bar{g}}|/|G|) \\ \textbf{subject to} \quad & 0 \leq \rho_\theta \leq 1 \\ & support_g \geq \rho_\theta \\ & G \in S/\theta \end{aligned} \tag{2}$$

, where $support_g = \frac{|S_g|}{|S/\theta|}$. To maximize the objective function, we utilize the *Hill Climbing* algorithm [29] to find the optimal support values. This in turn produces subgraphs of optimized quality.

⟨*description*⟩ ::= ⟨*sentence*⟩*

⟨*sentence*⟩ ::= ⟨*sentence*⟩ 'and' ⟨*sentence*⟩
| ⟨*statement*⟩ ⟨*modifier*⟩

⟨*statement*⟩ ::= ⟨*subject*⟩ ⟨*verb*⟩ ⟨*object*⟩

⟨*subject*⟩ ::= ⟨*noun phrase*⟩

⟨*object*⟩ ::= ⟨*noun phrase*⟩
| ⟨*empty*⟩

⟨*modifier*⟩ ::= ⟨*modifier*⟩ ⟨*conj*⟩ ⟨*modifier*⟩
| ⟨*when*⟩ ⟨*sentence*⟩
| ⟨*if*⟩ ['not'] ⟨*sentence*⟩
| ⟨*constant*⟩
| ⟨*empty*⟩

⟨*conj*⟩ ::= 'and'
| 'or'

⟨*when*⟩ ::= 'once'

⟨*if*⟩ ::= 'if'
| 'depending on if'

⟨*empty*⟩ ::= ' '

Figure 7: An Abbreviated Syntax of Our Descriptions

We follow the approach in the previous work [40] and conduct *concept learning* to obtain 109 security-sensitive APIs. Hence, we focus on these APIs and perform "API-oriented" behavior mining on 1000 randomly-collected top Android apps. More concretely, we first construct the subset, S/θ, specific to each individual API. On average, each subset contains 17 graphs. Then, we apply subgraph mining algorithm [38] to each subset.

Figure 6 exemplifies our mining process. Specifically, it shows that we discover a behavior pattern for the API `getLastKnownLocation()`. This pattern involves two other API calls, `getLongitude()` and `getLatitude()`. It demonstrates the common practice to retrieve location data in Android programs.

4.2 Graph Compression

Now that we have identified common subgraphs in the raw `SBG`s, we can further compress these raw graphs by replacing entire subgraphs with individual nodes. This involves two steps, subgraph isomorphism and subgraph collapse. We utilize the VF2 [13] algorithm to solve the subgraph isomorphism problem. In order to maximize the graph compression rate, we always prioritize a better match (i.e., larger subgraph). To perform subgraph collapse, we first replace subgraph nodes with one single new node. Then, we merge the attributes (i.e., context, conditions and constants) of all the removed nodes, and put the merged label onto the new one.

5. DESCRIPTION GENERATION

5.1 Automatically Generated Descriptions

Given a behavior graph `SBG`, we translate its semantics into textual descriptions. This descriptive language follows a subset of English grammar, illustrated in Figure 7 using Extended Backus-Naur form (EBNF). The `description` of an app is a conjunction of individual `sentence`s. An atomic `sentence` makes a `statement` and specifies a `modifier`. Recursively, a non-empty atomic `modifier` can be an adverb clause of condition, which contains another `sentence`.

The translation from a `SBG` to a textual description is then to map the graph components to the counterparts in this reduced language. To be more specific, each vertex of a graph is mapped to a single `sentence`, where the API or behavioral pattern is represented by a `statement`; the conditions, contexts and constant parameters are expressed using a `modifier`. Each edge is then translated to "`and`" to indicate data dependency.

One `sentence` may have several `modifier`s. This reflects the fact that one API call can be triggered in compound conditions and contexts, or a condition/context may accept several parameters. The modifiers are concatenated with "`and`" or "`or`" in order to verbalize specific logical relations. A context `modifier` begins with "`once`" to show the temporal precedence. A condition `modifier` starts with either "`if`" or "`depending on if`". The former is applied when a condition is statically resolvable while the latter is prepared for any other conservative cases. Notice that it is always possible to find more suitable expressions for these conjunctions.

In our motivating example, `getLine1Number()` is triggered under the condition that a specific button is selected. Due to the sophisticated internal computation, we did not extract the exact predicates. To be safe, we conservatively claim that the app retrieves the phone number *depending on if the user selects Button "Confirm"*.

5.2 Behavior Description Model

Once we have associated a behavior graph to this grammatical structure, we further need to translate an API operation or a pattern to a proper combination of subject, verb and object. This translation is realized using our *Behavior Description Model*. Conditions and contexts of `SBG`s are also translated using the same model because they are related to API calls.

We manually create this description model and currently support 306 sensitive APIs and 103 API patterns. Each entry of this model consists of an API or pattern signature and a 3-tuple of natural language words for subject, verb and object. We construct such a model by studying the Android documentation [6]. For instance, the Android API call `createFromPdu(byte[])` programmatically constructs incoming SMS messages from underlying raw Protocol Data Unit (PDU) and hence it is documented as "Create an SmsMessage from a raw PDU" by Google. Our model records its API prototype and assigns texts "the app", "retrieve" and "incoming SMS messages" to the three linguistic components respectively. These three components form a sentence template. Then, constants, concrete conditions and contexts serve as modifiers to complete the template. For example, the template of `HttpClient.execute()` is represented using words "the app", "send" and "data to network". Suppose an app uses this API to deliver data to a constant URL "http://constant.url", when the phone is locked (i.e., keyguard is on). Then, such constant value and condition will be fed into the template to produce the sentence "*The app sends data to network "http://constant.url" if the phone is locked.*" The condition APIs share the same model format. The API checking keyguard status (i.e., `KeyguardManager.isKeyguardLocked()`) is modeled as words "the phone", "be" and "locked".

It is noteworthy that an alternative approach is to generate this model programmatically. Sridhara et al. [31] proposed to automatically extract descriptive texts for APIs and produce the Software Word Usage Model. The API name, parameter type and return type are examined to extract the linguistic elements. For example, the model of `createFromPdu(byte[])` may therefore contain the keywords "create", "from" and "pdu", all derived from the function name. Essentially, we can take the same approach. However, we argue that such a generic model was designed to assist software development and is not the best solution to our problem. An average user may not be knowledgeable enough to understand the low-level technical terms, such as "pdu". In contrast, our text selections (i.e., "the app", "retrieve" and "incoming SMS messages") directly explain the behavior-level meaning.

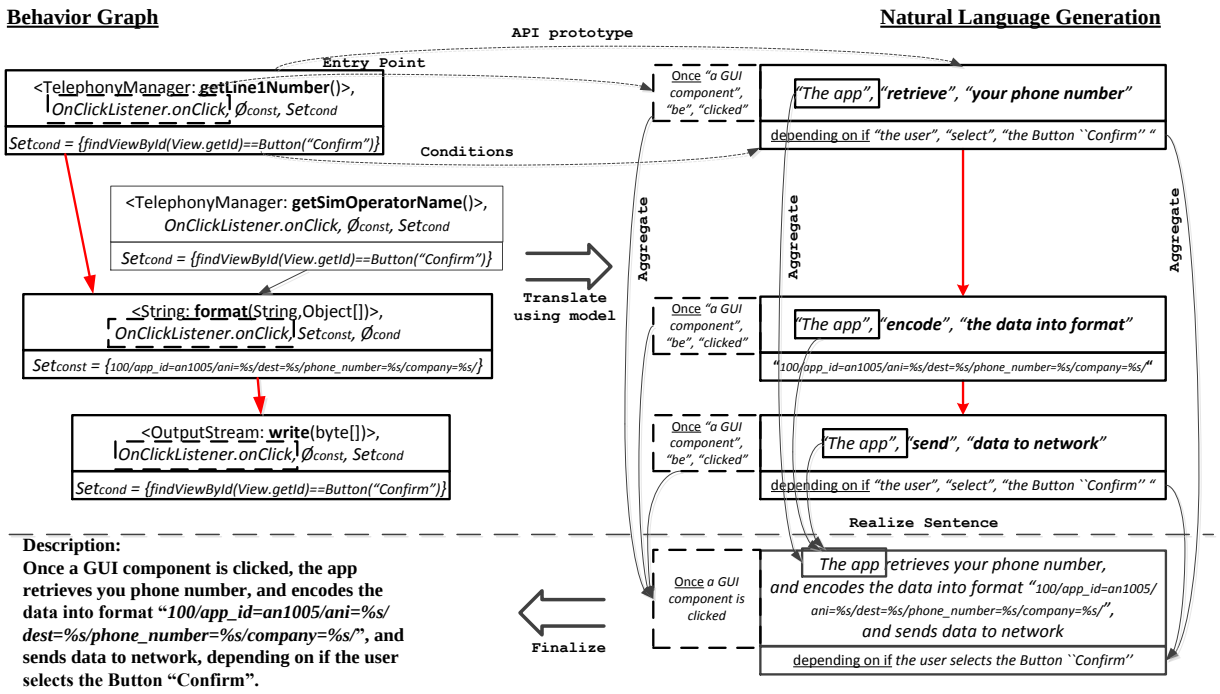


Figure 8: Description Generation for the Motivating Example

Table 1: Program Logics in Behavioral Patterns

Program Logic	How to Describe
Singleton Retrieval	Describe the latter.
Workflow	Describe both.
Access to Hierarchical Data	Describe the former.

We generate description model for API patterns based on their internal program logics. Table 1 presents the three major logics that we have discovered in behavioral patterns. 1) A singleton object is retrieved for further operations. For example, a `SmsManager.getDefault()` is always called prior to `SmsManager.sendTextMessage()` because the former fetches the default `SmsManager` that the latter needs. We therefore describe only the latter which is associated to a more concrete behavior. 2) Successive APIs constitute a dedicated workflow. For instance, `divideMessage()` always happens before `sendMultipartTextMessage()`, since the first provides the second with necessary inputs. In this case, we study the document of each API and describe the complete behavior as an entirety. 3) Hierarchical information is accessed using multiple levels of APIs. For instance, to use location data, one has to first call `getLastKnownLocation()` to fetch a `Location` object, and then call `getLongitude()` and `getLatitude()` to read the "double"-typed data from this object. Since the higher level object is already meaningful enough, we hence describe this whole behavior according to only the former API.

In fact, we only create description models for 103 patterns out of the total 109 discovered ones. Some patterns are large and complex, and are hard to summarize. For these patterns, we have to fall back to the safe area and describe them in a API-by-API manner.

In order to guarantee the security-sensitivity and readability of the descriptive texts, we carefully select the words to accommodate the model. To this end, we learn from the experience of prior security studies [26,28] on app descriptions: 1) The selected vocabulary must be straightforward and stick to the essential API functionalities. As an counterexample, an audio recording behavior can hardly be inferred from the script "*Blow into the mic to extinguish the flame like a real candle*" [26]. This is because it does not explicitly refer to the audio operation. 2) Descriptive texts must be distinguishable for semantically different APIs. Otherwise, poorly-chosen texts may confuse the readers. For instance, an app with description "*You can now turn recordings into ringtones*" in reality only converts previously recorded files to ringtones, but can be mistakenly associated to the permission `android.permission.RECORD_AUDIO` due to the misleading text choice [26, 28].

Notice that the model generation is a one-time effort. Moreover, this manual effort is a manageable process due to two reasons. First, we exclusively focus on security-sensitive behaviors and therefore describe only security-related APIs. After applying *concept learning*, we further conclude that, a limited amount of sensitive APIs contributes to a majority of harmful operations. Thus, we can concentrate on and create models for more crucial ones. Second, the number of discovered patterns is also finite. This is because we can tune the parameters of objective function (Equation 2) so that the amount of identified subgraphs is manageable.

5.3 Behavior Graph Translation

Now that we have defined a target language and prepared a model to verbalize sensitive APIs and patterns, we further would like to translate an entire behavior graph into natural language scripts. Algorithm 2 demonstrates our graph traversal based translation.

This algorithm takes a `SBG` G and the description model M_{desc} as the inputs and eventually outputs a set of descriptions. The overall idea is to traverse the graph and translate each path. Hence, it first performs a breadth-first search and collects all the paths into Set_{path}. Notice that the graph traversal algorithm (i.e., BFS or DFS) does not affect the quality of output. Next, it examines each path in Set_{path} to parse the nodes in sequence. Each node is then parsed to extract the node name, constants, conditions and contexts. The node name $node.name$ (API or pattern) is used to query the model M_{desc} and fetch the {subj,vb,obj} of a main clause. The

Algorithm 2 Generating Descriptions from a SBG

$G \leftarrow$ {A SBG }
$M_{desc} \leftarrow$ {Description model}
$Set_{desc} \leftarrow \emptyset$
$Set_{path} \leftarrow$ BFS(G)
for $path \in Set_{path}$ **do**
  $desc \leftarrow$ null
  for $node \in path$ **do**
    {subj,vb,obj} $\leftarrow$ Query$_{M_{desc}}(node.name)$
    Cmod $\leftarrow$ null
    $Set_{const} \leftarrow$ GetConsts($node$)
    for $\forall const \in Set_{const}$ **do**
      Cmod $\leftarrow$ Aggregate(Cmod,$const$)
    end for
    $Set_{cc} \leftarrow$ GetConditionsAndContext($node$)
    for $\forall cc \in Set_{cc}$ **do**
      {subj,vb,obj}$_{cc} \leftarrow$ Query$_{M_{desc}}(cc)$
      $text_{cc} \leftarrow$ RealizeSentence({subj,vb,obj}$_{cc}$)
      Cmod $\leftarrow$ Aggregate(Cmod,$text_{cc}$)
    end for
    $text \leftarrow$ RealizeSentence({subj,vb,obj,Cmod})
    $desc \leftarrow$ Aggregate($desc, text$)
  end for
  $Set_{desc} \leftarrow Set_{desc} \cup \{desc\}$
end for
output Set_{desc} as the generated description set

constants, conditions and contexts are organized into the modifier (Cmod) of main clause, respectively. In the end, the main clause is realized by assembling {subj,vb,obj} and the aggregate modifier Cmod. The realized sentence is inserted into the output set Set_{desc} if it is not a redundant one.

5.4 Motivating Example

We have implemented the natural language generation using a NLG engine [7] in 3K LOC. Figure 8 illustrates how we step-by-step generate descriptions for the motivating example.

First, we discover two paths in the SBG: 1) `getLine1Number()` → `format()` → `write()` and 2) `getSimOperatorName()` → `format()` → `write()`.

Next, we describe every node sequentially on each path. For example, for the first node, the API `getLine1Number()` is modeled by the 3-tuple {"the app", "retrieve", "your phone number"}; the entry point `OnClickListener.onClick` is mapped to {"a GUI component", "be", "clicked"} and preceded by "Once"; the condition `findViewById(View.getId)==Button("Confirm")` is translated using the template {"the user", "select", " "}, which accepts the GUI name, *Button "Confirm"*, as a parameter. The condition and main clause are connected using "depending on if".

At last, we aggregate the sentences derived from individual nodes. In this example, all the nodes share the same entry point. Thus, we only keep one copy of *"Once a GUI component is clicked"*. Similarly, the statements on the nodes are also aggregated and thus share the same subject "The app". We also aggregate the conditions in order to avoid the redundancy. As a result, we obtain the description illustrated at the bottom left of Figure 8.

6. EVALUATION

In this section, we evaluate the correctness, effectiveness, conciseness of generated descriptions and the runtime performance of DESCRIBEME.

Table 2: Description Generation Results for DroidBench

Total #	Correct	Missing Desc.	False Statement
65	55	6	4

6.1 Correctness and Security-Awareness

Correctness. To evaluate the correctness, we produce textual descriptions for DroidBench apps (version 1.1) [3]. DroidBench apps are designed to assess the accuracy of static analyses on Android programs. We use these apps as the ground truths because they are open-sourced programs with clear semantics. However, it is worth noting that DroidBench does not include any test cases for native code or dynamic loaded classes. Thus, this evaluation only demonstrates whether DESCRIBEME can correctly discover the static program behaviors at bytecode level. In fact, static analysis in general lacks the capability of extracting runtime behaviors and can be evaded accordingly. Nevertheless, we argue that any analysis tools, both static and dynamic, can be utilized in our framework to achieve the goal. Detailed discussion is presented in Section 7.1.

Table 2 presents the experimental results, which show that DESCRIBEME achieves a true positive rate of 85%. DESCRIBEME misses behavior descriptions due to three major reasons. 1) Points-to analysis lacks accuracy. We rely on Soot's capability to perform points-to analysis. However, it is not precise enough to handle the instance fields accessed in callback functions. 2) DESCRIBEME does not process exception handler code and therefore loses track of its dataflow. 3) Some reflective calls cannot be statically resolved. Thus, DESCRIBEME fails to extract their semantics.

DESCRIBEME produces false statements mainly because of two reasons. First, our static analysis is not sensitive to individual array elements. Thus, it generates false descriptions for the apps that intentionally manipulate data in array . Second, again, our points-to analysis is not accurate and may lead to over-approximation.

Despite the incorrect cases, the accuracy of our static analysis is still comparable to that of FlowDroid [9], which is the state-of-the-art static analysis technique for Android apps. Moreover, we would like to again point out that the accuracy of static analysis is not the major focus of this work. Our main contribution lies in the fact that, we combine program analysis with natural language generation so that we can automatically explain program behaviors to end users in human language.

Permission Fidelity. To demonstrate the security-awareness of DESCRIBEME, we use a description vetting tool, AutoCog [28], to evaluate the "permission-fidelity" of descriptions. AutoCog examines the descriptions and permissions of an app to discover their discrepancies. We use it to analyze both the original descriptions and the security-centric ones produced by DESCRIBEME, and assess whether our descriptions can be associated to more permissions that are actually requested.

Unfortunately, AutoCog only supports 11 permissions in its current implementation. In particular, it does not handle some crucial permissions that are related to information stealing (e.g., phone number, device identifier, service provider, etc.), sending and receiving text messages, network I/O and critical system-level behaviors (e.g., `KILL_BACKGROUND_PROCESSES`). The limitation of AutoCog in fact brings difficulties to our evaluation: if generated descriptions are associated to these unsupported permissions, AutoCog fails to recognize them and thus cannot conduct equitable assessment. Such a shortcoming is also shared by another NLP-based (i.e., natural language processing) vetting tool, WHYPER [26], which focuses on even fewer (3) permissions. This implies that it is a major challenge for NLP-based approaches to achieve high permission coverage, probably because it is hard to correlate texts to semantically obscure permissions (e.g., `READ_PHONE_STATE`). In

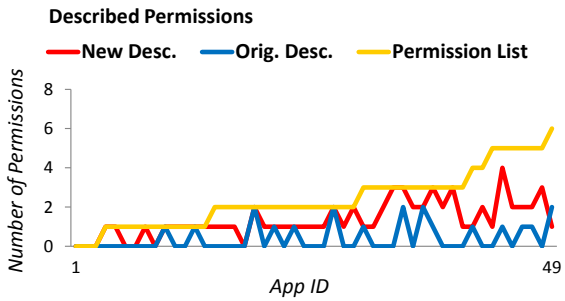


Figure 9: Permissions Reflected in Descriptions

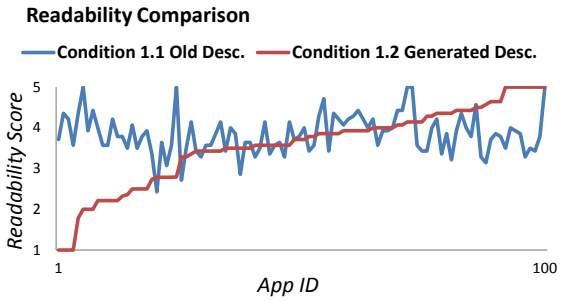


Figure 10: Readability Ratings

contrast, our approach does not suffer from this limitation because API calls are clearly associated to permissions [10].

Despite the difficulties, we manage to collect 30 benign apps from Google play and 20 malware samples from Malware Genome Project [5], whose permissions are supported by AutoCog. We run DESCRIBEME to create the security-centric descriptions and present both the original and generated ones to AutoCog. However, we notice that AutoCog sometimes cannot recognize certain words that have strong security implications. For example, DESCRIBEME uses "*geographic location*" to describe the permissions `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. Yet, AutoCog cannot associate this phrase to any of the permissions.

The fundamental reason is that AutoCog and DESCRIBEME use different glossaries. AutoCog performs machine learning on a particular set of apps and extracts the permission-related glossary from these existing descriptions. In contrast, We manually select descriptive words for each sensitive API, using domain knowledge.

To bridge this gap, we enhance AutoCog to recognize the manually chosen keywords. The experimental result is illustrated in Figure 9, where X-axis is the app ID and Y-axis is the amount of permissions. The three curves, from top to bottom, represent the amounts of permissions that are requested by the apps, recognized by AutoCog from security-centric descriptions and identified from original descriptions, respectively. Cumulatively, 118 permissions are requested by these 50 apps. 20 permissions are discovered from the old descriptions, while 66 are uncovered from our scripts. This reveals that DESCRIBEME can produce descriptions that are more security-sensitive than the original ones.

DESCRIBEME fails to describe certain permission requests due to three reasons. First, some permissions are used for native code or reflections that cannot be resolved. Second, a few permissions are not associated to API calls (e.g., `RECEIVE_BOOT_COMPLETED`), and thus are not included into the `SBG`s. Last, some permissions are correlated to certain API parameters. For instance, the `query` API requires permission `READ_CONTACTS` only if the target URI is the `Contacts` database. Thus, if the parameter value cannot be extracted statically, such a behavior will not be described.

6.2 Readability and Effectiveness

To evaluate the readability and effectiveness of generated descriptions, we perform a user study on the Amazon's Mechanical Turk (MTurk) [1] platform. The goal is two-fold. First, we hope to know whether the generated scripts are readable to average audience. Second, we expect to see whether our descriptions can actually help users avoid risky apps. To this end, we follow Felt et al.'s approach [16], which also designs experiments to understand the impact of text-based protection mechanisms.

Methodology. We produce the security-centric descriptions for Android apps using DESCRIBEME and measure user reaction to the old descriptions (Condition 1.1, 2.1-2.3), machine-generated ones (Condition 2.1) and the *new descriptions* (Condition 2.4-2.6). Notice that the *new description* is the old plus the generated one.

Dataset. Due to the efficiency consideration, we perform the user study based on the descriptions of 100 apps. We choose these 100 apps in a mostly random manner but we also consider the distribution of app behaviors. In particular, 40 apps are malware and the others are benign. We manually inspect the 60 benign ones and further put them into two categories: 16 privacy-breaching apps and 44 completely clean ones.

Participants Recruitment. We recruit participants directly from MTurk and we require participants to be smartphone users. We also ask screening questions to make sure participants understand basic smartphone terms, such as "Contacts" or "GPS location".

Hypotheses and Conditions.

Hypothesis 1: Machine-generated descriptions are readable to average smartphone users. To assess the readability, we prepare both the old descriptions (Condition 1.1) and generated ones (Condition 1.2) of the same apps. We would like to evaluate machine-generated descriptive texts via comparison.

Hypothesis 2: Security-centric descriptions can help reduce the downloading of risky apps. To test the impact of the security-centric descriptions, we present both the old and *new* (i.e., old + generated) descriptions for malware (Condition 2.1 and 2.4), benign apps that leak privacy (Condition 2.2 and 2.5) and benign apps without privacy violations (Condition 2.3 and 2.6). We expect to assess the app download rates on different conditions.

User Study Deployment. We post all the descriptions on MTurk and anonymize their sources. We inform the participants that the tasks are about Android app descriptions and we pay 0.3 dollars for each task. Participants take part in two sets of experiments. First, they are given a random mixture of original and machine-generated descriptions, and are asked to provide a rating for each script with respect to its readability. The rating is ranged from 1 to 5, where 1 means completely unreadable and 5 means highly readable.

Second, we present the participants another random sequence of descriptions. Such a sequence contains both the old and *new* descriptions for the same apps. Again, we stress that the *new description* is the old one plus the generated one. Then, we ask participants the following question: "*Will you download an app based on the given description and the security concern it may bring to you?*". We emphasize "security concern" here and we hope participants should not accept or reject an app due to the considerations (e.g., functionalities, personal interests) other than security risks.

Limitations. The security-centric descriptions are designed to be the supplement to the original ones. Therefore, we present the two of them as an entirety (i.e., *new description*) to the audience, in the second experiment. However, this may increase the chance for participants to discover the correlation between a pair of old and new descriptions. As a result, we introduce randomness into the display order of descriptions to mitigate the possible impact.

Results and Implications. Eventually, we receive 573 responses

Table 3: App Download Rates (ADR)

#	Condition	ADR
2.1	Malware w/ old desc.	63.4%
2.2	Leakage w/ old desc.	80.0%
2.3	Clean w/ old desc.	71.1%
2.4	Malware w/ new desc.	24.7%
2.5	Leakage w/ new desc.	28.2%
2.6	Clean w/ new desc.	59.3%

and a total of 2865 ratings. Figure 10 shows the readability ratings of 100 apps for Condition 1.1 and 1.2. For our automatically created descriptions, the average readability rating is 3.596 while over 80% readers give a rating higher than 3. As a comparison, the average rating of the original ones is 3.788. This indicates our description is readable, even compared to texts created by human developers. The figure also reveals that the readability of human descriptions are relatively stable while machine-generated ones sometimes bear low ratings. In a further investigation, we notice that our descriptions with low ratings usually include relatively technical terms (e.g., subscriber ID) or lengthy constant string parameters. We believe that this can be further improved during post-processing. We discuss this in Section 7.2.

Table 3 depicts experimental results for Condition 2.1 - 2.6. It demonstrates the security impact of our new descriptions. We can see a 38.7% decrease of application download rate (ADR) for malware, when the new descriptions instead of old ones are presented to the participants. We believe that this is because malware authors deliberately provide fake descriptions to avoid alerting victims, while our descriptions can inform users of the real risks. Similar results are also observed for privacy-breaching benign apps, whose original descriptions are not focused on the security and privacy aspects. On the contrary, our descriptions have much less impact on the ADR of clean apps. Nevertheless, they still raise false alarms for 11.8% participants. We notice that these false alarms result from descriptions of legitimate but sensitive functionalities, such as accessing and sending location data in social apps. A possible solution to this problem is to leverage the "peer voting" mechanism from prior work [23] to identify and thus avoid documenting the typical benign app behaviors.

6.3 Effectiveness of Behavior Mining

Next, we evaluate the effectiveness of behavior mining. In general, we have discovered 109 significant behaviors involving 109 sensitive APIs, via subgraph mining in 2069 SBGs of 1000 Android apps. Figure 11 illustrates the sizes of the identified subgraphs and shows that one subgraph contains 3 nodes on average. We further study these pattern graphs. As presented in Table 1, they effectively reflect common program logics and programming conventions.

Furthermore, we reveal that the optimal patterns of different APIs are extracted using distinctive support threshold values. Figure 12 depicts the distribution of selected support thresholds over 109 APIs. It indicates that a uniform threshold cannot guarantee to produce satisfying behavior pattern for every API. This serves as a justification for our "API-oriented" behavior mining.

To show the reduction of description size due to behavior mining, we compare the description sizes of raw SBGs and compressed ones. We thus produce descriptions for 235 randomly chosen apps, before and after graph compression. The result, illustrated in Figure 13, depicts that for over 32% of the apps, the scripts derived from compressed graphs are shorter. The maximum reduction ratio reaches 75%. This indicates that behavior mining effectively helps produce concise descriptions.

6.4 Runtime Performance

We evaluate the runtime performance for 2851 apps. Static program analysis dominates the runtime, while the description generation is usually fairly fast (under 2 seconds). The average static analysis runtime is 391.5 seconds, while the analysis for a majority (80%) of apps can be completed within 10 minutes. In addition, almost all the apps (96%) are processed within 25 minutes. Notice that, though it may take minutes to generate behavior graphs, this is a one-time effort, for a single version of each app. Provided there exists a higher requirement on analysis latency, we can alternatively seek more speedy solutions, such as AppAudit [35].

7. DISCUSSION

7.1 Evasion

The current implementation of DESCRIBEME relies on static program analysis to extract behavior graphs from Android bytecode programs. However, bytecode-level static analysis can cause false negatives due to two reasons. First, it cannot cope with the usage of native code as well as JavaScript/HTML5-based programs running in WebView. Second, it cannot address the dynamic features of Android programs, such as Java reflection and dynamic class loading. Thus, any critical functionalities implemented using these techniques can evade the analysis in DESCRIBEME.

Even worse is that both benign and malicious app authors can intentionally obfuscate their programs, via Android packers [2,4,42], in order to defeat static analysis. Such packers combine multiple dynamic features to hide real bytecode program, and only unpack and execute the code at runtime. As a result, DESCRIBEME is not able to extract the true behaviors from packed apps.

However, we argue that the capability of analysis technique is orthogonal to our main research focus. In fact, any advanced analysis tools can be plugged into our description generation framework. In particular, emulation-based dynamic analysis, such as CopperDroid [33] or DroidScope [37], can capture the system-call level runtime behaviors and therefore can help enable the description of the dynamic features; symbolic execution, such as AppIntent [39], can facilitate the solving of complex conditions.

7.2 Improvement of Readability

There exists room to improve the readability of automatically generated descriptions. In fact, some of the raw text is still technical to the average users. We hope higher readability can be achieved by post-processing the generated raw descriptions. That is, we may combine natural language processing (NLP) and natural language generation (NLG) techniques to automatically interpret the "raw" text, select more appropriate vocabulary, re-organize the sentence structure in a more smooth manner and finally synthesize a more natural script. We may also introduce experts' knowledge or crowd-sourcing and leverage an interactive process to gradually refine the raw text.

8. RELATED WORK

Software Description Generation. There exists a series of studies on software description generation for traditional Java programs. Sridhara et al. [30] automatically summarized method syntax and function logic using natural language. Later, they [32] improved the method summaries by also describing the specific roles of method parameters. Further, they [31] automatically identified high-level abstractions of actions in code and described them in natural language.In the meantime, Buse [11] leveraged symbolic execution and code summarization technique to document program differences. Moreno et al. [25] proposed to discover class and method stereotypes and use such information to summarize Java classes.The goal of these studies is to improve the program comprehension for

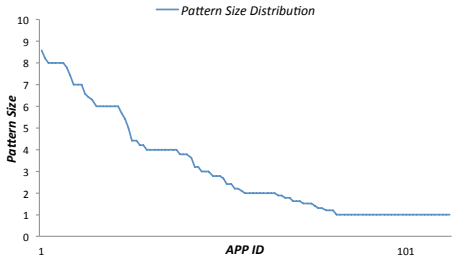


Figure 11: Subgraph Sizes

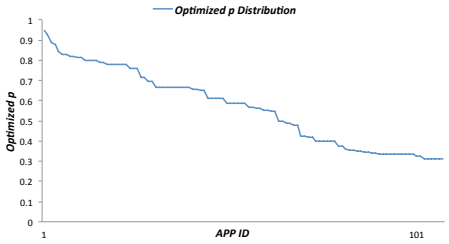


Figure 12: Optimal Support Thresholds

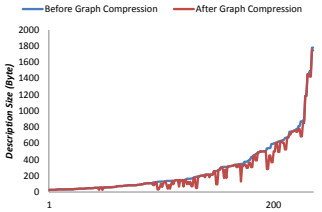


Figure 13: Size Reductions

developers. As a result, they focus on documenting intra-procedural program logic and low-level code structures. On the contrary, DESCRIBEME aims at helping end users to understand the risk of Android apps, and therefore describes high-level program semantics.

Text Analytics for Android Security. Recently, efforts have been made to study the security implications of textual descriptions for Android apps. WHYPER [26] used natural language processing technique to identify the descriptive sentences that are associated to permissions requests. AutoCog [28] further applied machine learning technique to automatically correlate the descriptive scripts to permissions. Inspired by these studies, we expect to automatically bridge the gap between the textual description and security-related program semantics.

Program Analysis using Graphs. Prior studies have focused on using behavior graphs for program analysis. Kolbitsch et al. [22] utilized dynamic analysis to extract syscall dependency graphs as signature, so as to discover unknown malicious programs. Fredrikson et al. [19] proposed an automated technique to extract near-optimal specifications that uniquely identify a malware family. Yamaguchi et al. [36] introduced the code property graph, which can model common vulnerabilities. Feng et al. [18] constructed kernel object graph for robust memory analysis. Zhang et al. [41] generated static taint graphs to help mitigate component hijacking vulnerabilities in Android apps. As a comparison, we take a step further and transform behavior graphs into natural language.

9. CONCLUSION

We propose a novel technique to automatically generate security-centric app descriptions, based on program analysis. We implement a prototype, DESCRIBEME, and evaluate our system using DroidBench and real-world Android apps. Experimental results demonstrate that DESCRIBEME can effectively bridge the gap between descriptions and permissions.

10. ACKNOWLEDGMENT

We would like to thank anonymous reviewers and our shepherd, Prof. Lorenzo Cavallaro, for their feedback in finalizing this paper. This research was supported in part by National Science Foundation Grant #1054605 and Air Force Research Lab Grant #FA8750-15-2-0106. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

11. REFERENCES

[1] amazon mechanical turk. https://www.mturk.com/mturk/welcome.

[2] bangcle. http://www.bangcle.com.

[3] Droidbench-benchmarks. http://sseblog.ec-spride.de/tools/droidbench/.

[4] ijiami. http://www.ijiami.cn.

[5] Malware Genome Project. http://www.malgenomeproject.org.

[6] Reference - Android Developers. http://developer.android.com/reference/packages.html.

[7] simplenlg: Java API for Natural Language Generation. https://code.google.com/p/simplenlg/.

[8] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[9] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)* (June 2014).

[10] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)* (October 2012).

[11] BUSE, R. P., AND WEIMER, W. R. Automatically Documenting Program Changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (September 2010).

[12] CHEN, K. Z., JOHNSON, N., D'SILVA, V., DAI, S., MACNAMARA, K., MAGRINO, T., WU, E. X., RINARD, M., AND SONG, D. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)* (February 2013).

[13] CORDELLA, L. P., FOGGIA, P., SANSONE, C., AND VENTO, M. A (Sub) Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2004).

[14] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)* (October 2010).

[15] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS'12)* (July 2012).

[16] FELT, A. P., REEDER, R. W., ALMUHIMEDI, H., AND CONSOLVO, S. Experimenting at Scale with Google Chrome's SSL Warning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'14)* (April 2014).

[17] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium* (August 2011).

[18] FENG, Q., PRAKASH, A., YIN, H., AND LIN, Z. MACE: High-Coverage and Robust Memory Analysis for Commodity Operating Systems. In *Proceedings of Annual Computer Security Applications Conference (ACSAC'14)* (December 2014).

[19] FREDRIKSON, M., JHA, S., CHRISTODORESCU, M., SAILER, R., AND YAN, X. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland'10)* (May 2010).

[20] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)* (February 2012).

[21] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (May 2014).

[22] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium* (August 2009).

[23] LU, K., LI, Z., KEMERLIS, V., WU, Z., LU, L., ZHENG, C., QIAN, Z., LEE, W., AND JIANG, G. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS'15)* (February 2015).

[24] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)* (October 2012).

[25] MORENO, L., APONTE, J., SRIDHARA, G., MARCUS, A., POLLOCK, L., AND VIJAY-SHANKER, K. Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the 2013 IEEE 21th International Conference on Program Comprehension (ICPC'13)* (May 2013).

[26] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Conference on Security* (August 2013).

[27] POYNTON, C. *Digital video and HD: Algorithms and Interfaces*. Elsevier, 2012.

[28] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 21st Conference on Computer and Communications Security (CCS)* (November 2014).

[29] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*. 2003.

[30] SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L., AND VIJAY-SHANKER, K. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (September 2010).

[31] SRIDHARA, G., POLLOCK, L., AND VIJAY-SHANKER, K. Automatically Detecting and Describing High Level Actions Within Methods. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)* (May 2011).

[32] SRIDHARA, G., POLLOCK, L., AND VIJAY-SHANKER, K. Generating Parameter Comments and Integrating with Method Summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC'11)* (June 2011).

[33] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)* (February 2015).

[34] WEI, F., ROY, S., OU, X., AND ROBBY. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)* (November 2014).

[35] XIA, M., GONG, L., LV, Y., QI, Z., AND LIU, X. Effective Real-time Android Application Auditing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland'15)* (May 2015).

[36] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland'14)* (May 2014).

[37] YAN, L.-K., AND YIN, H. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium* (August 2012).

[38] YAN, X., AND HAN, J. gspan: Graph-based Substructure Pattern Mining. In *Proceedings of IEEE International Conference on Data Mining(ICDM'03)* (December 2002).

[39] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)* (November 2013).

[40] ZHANG, M., DUAN, Y., YIN, H., AND ZHAO, Z. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)* (November 2014).

[41] ZHANG, M., AND YIN, H. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)* (February 2014).

[42] ZHANG, Y., LUO, X., AND YIN, H. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)* (September 2015).

[43] ZHOU, Y., AND JIANG, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)* (May 2012).

[44] ZHOU, Y., AND JIANG, X. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)* (February 2013).

[45] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (February 2012).

[46] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11)* (June 2011).

APPENDIX

A. SECURITY-CENTRIC DESCRIPTIONS OF THE MOTIVATING EXAMPLE

Once a GUI component is clicked, the app reads data from network and sends data to network, depending on if the user selects Button "Confirm".

Once a GUI component is clicked, the app retrieves you phone number, and econdes the data into format "100/app_id=an1005/ani=%s/dest=%s/phone_number=%s/company=%s/", and sends data to network, depending on if the user selects the Button "Confirm".

Once a GUI component is clicked, the app retrieves the service provider name, and econdes the data into format "100/app_id=an1005/ani=%s/dest=%s/phone_number=%s/company=%s/", and sends data to network, depending on if the user selects the Button "Confirm".

The app retrieves text from user input and displays text to the user.

Once a GUI component is clicked, the app retrieves text from user input and sends data to network, depending on if the user selects Button "Confirm".

The app opens a web page.

The app reads from file "address.txt".

The app reads from file "contact.txt".

The app reads from file "message.txt".