Turn Words into Actions: A Natural Language Interface to Programs for App Customization

Abstract

App customization modifies code and enforces security policies. General app customization is limited to analysts with particular domain knowledge on app security analysis. In this paper, we raise a question that is it possible to have a better user interface for everyone to programs? To address this problem, we present an approach towards a natural language interface to programs for app customization. We build an interface prototype, named *IronDroid*, to customize apps according to natural language inputs. Our approach reduces the semantic gap from general human languages to specific app customization policies. We provide an end-to-end design from natural languages to automated app customization. We transform natural languages into structure and dependence trees. Program inputs are extracted by mining the transformed trees. We generate rewriting specifications via a user-intention-guided taint flow analysis. The app is rewritten by following rewriting specifications using our rewriting framework.

Our work showcases a new application of app customization with a more natural user interface. Although understanding languages for generating security-oriented rewriting policies is challenging, our attempt shows a promising opportunity to generate a natural language interface for app customization. In the experiment, our approach achieves 90.2% accuracy in understanding user intentions. Our approach successfully rewrites both benchmark and real-world apps. It incurs negligible performance overhead in rewriting (3.3%). We also apply *IronDroid* for real-world security customization scenarios.

1. INTRODUCTION

The popularity of Android apps introduces new opportunities and challenges in security research. Existing research aims at detecting malware [5, 39, 43, 6, 29, 15, 41]. These approaches screen malware with reasonable success. However, these approaches are designed for general-purpose malware detection. They are not targeted to modify apps or restrict app behaviors. The post-detection customization is necessary for app security enforcement. Users utilize app customization to protect privacy and enhance security. In this paper, we consider the problem of app customization by rewriting apps' bytecode. We explore the possibilities to have a better user interface for everyone to programs. We motivate our approach to enlarge the popularity of users who can use app customization techniques. Our goal is to provide a more user friendly interface for app customization.

We motivate the need of app customization in three aspects: *1) Customized security preferences.* Users have individual privacy preferences on the personal data. The number of people who do not trust mobile apps is growing rapidly[1]. A recent survey suggests that 54% of mobile users are worried about personal privacy[2]. Users would like to customize apps for individual privacy preferences. *2) Economic reasons.* Legitimate apps may potentially snoop on users' private data and track their activities[3]. Untrusted libraries are capable of collecting information without user awareness[4]. Users would like to customize apps for better security and privacy. *3) Transparency of app logic.* Users are unaware of how their data is used and transformed. Sensitive data could be acquired and transmitted in multiple ways (e.g., to other apps or the internet). Apps contain complex code structures and multiple components (e.g., activities and services), and generally users do not know the inner working of the apps they install.

These factors push the need for app customization. Users have motivations to customize apps and restrict private data usage. A user friendly interface could significantly facilitate the app customization process. Security analysts and normal users benefit from the interface for generating security-oriented customization policies. The *technical challenge* to build a natural language interface to programs is the semantic gap between language-level representations and program-level specifications. Google makes a recent effort with Android dynamic permission [2]. Dynamic permission allows users to deny access to private data on the system level. It only provides limited operations: turning access on or off. General app rewriting solutions [12, 18] are based on straightforward function parsing. They are neither capable of extracting user intentions nor feasible for complex customization requirements. Current language-based control systems support functionalities for particular tasks, e.g.,

[1]https://goo.gl/gtFDu7
[2]https://goo.gl/RlSTzA
[3]https://goo.gl/KOV0Al
[4]https://goo.gl/6vJDee

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

opening a door or turning on a TV. However, security-oriented app customization requires the modification of apps to match domain specific requirements. These domain specific requirements are generally generated by a limited number of security analysts or experts. How to reduce the semantic gap is still unclear in the existing solutions.

Our goal is to provide a natural language interface to programs for app customization. A naive approach is to ask users to complete questionnaires and surveys. Security specifications are extracted by experienced experts from these questionnaires and surveys. This approach needs manual verification and is not scalable for general app customization. It is tedious and error-prone to ask users for security policies. Inspired by recent achievements in language understanding, our approach identifies user intentions from natural languages. We aim to reduce the *semantic gap* from natural languages to rewriting specifications. The semantic gap is that language-level sentences and code-level rewriting contain different structures and representations. Our main technical challenge is to eliminate the semantic gap between the two abstractions of representations.

In this paper, we present *IronDroid*[5], a new app customization tool with a natural language interface. *IronDroid* enables automatic app rewriting with security-oriented natural language analysis. We propose a new mining algorithm to extract user security preferences from natural languages. The rewriting specification is extracted via a user-intention-guided taint flow analysis. The rewriting specification focuses on the sensitive sinks that result in privacy violation. The app is automatically modified by following rewriting specifications. In our design, we provide an efficient interface for language-level and code-level representation transition, blazing the trail toward usable app customization for a more usable rewriting solution. We also demonstrate a general rewriting framework which supports complex app modifications. We summarize our contributions as follows:

- We study the problem a natural language interface in the app customization. We present an interface design for customizing apps' behaviors based on natural languages. Our prototype *IronDroid* extracts user intentions from natural languages and customizes apps appropriately. *IronDroid* efficiently transforms natural language descriptions into rewriting specifications.
- We provide a new graph-mining-based algorithm to identify user intentions from natural languages. We propose a user-intention-based taint flow analysis for mapping natural languages into rewriting specifications. We build a general rewriting framework that is feasible for complex app customization.
- We implemented an automatic rewriting framework for app customization. In the experiment, our approach achieves 90.2% in understanding user intentions, a significant improvement over the state-of-the-art solution. Our framework supports more rewriting operations than existing solutions. Our rewriting successfully rewrites all benchmark apps. Our approach introduces negligible 3.3% overhead in rewriting. We also extend *IronDroid* for practical privacy protection in real-world apps.

In this paper, we explore the possibilities for a natural language interface for app customization. Our work presents an enabling technology to enhance app security with a natural language interface. Although understanding languages for generating security-oriented rewriting policies is challenging, we make impressive progresses toward automatic app customization. Our design includes natural language transition, rewriting specification extraction, user-intention-guided guided taint flow analysis and automatic app rewriting. Our approach presents the first attempt for natural language supported app customization. The integration of app customization with natural language processing would significantly enlarge the popularity to apply security-oriented app customization techniques. The interface would allow users to protect privacy and enhance security more easily.

We point out the challenges for processing language sentences to rewrite specifications. We provide several possible mitigations to improve the practical usability of the prototype. *IronDroid* is able to restrict and rewrite app behaviors for security purposes. A more advanced version of our approach could be embedded into voice control assistants (e.g., Siri [3], Alexa [1]). However, before natural language analysis and taint flow analysis can be made fully reliable and usable, the use scenario is unlikely.

2. OVERVIEW

In this section, we present security applications of rewriting, our assumption, technical challenges and the definitions needed to understand our approach.

2.1 Technical Challenges And Assumption

Security applications of rewriting. We summarize two rewriting scenarios for security.

Policy Enforcement. Organization administrators could use rewriting to enforce organization security policies. The security policies can be customized to meet organization requirements. A natural language interface reduces manual efforts to define security policies.

Privacy Protection. Users could use rewriting to protect privacy. A natural language interface provides possibilities for a user to specify her personal security concern. The interface rewrites the app to satisfy a user's expectations.

Goal. In this paper, our goal is to design a natural language interface to programs for rewriting. Although accurately understanding natural languages is challenging in practice, we elaborate our efforts towards applying languages processing in security-oriented rewriting. the interface enlarges the population (e.g., normal users and analysts) to adopt app customization for security. The interface transforms a language sentence into program inputs (i.e., sources and sinks) for generating rewriting specifications. Our current rewriting specifications only focus on the sinks with privacy violation. We rewrite sensitive sinks to enhance security of an app.

Assumption. We assume apps that pass the vetting process are still not fully trusted. The assumption is reasonable because a recent study found hundreds of untrusted apps in the official market[6]. Users have privacy concerns on the

[5] *IronDroid* is inspired by the movie IronMan. We aim to automatically customize an app given an user's natural language input. The app is rewritten by enforcing user-defined security policies.

[6] https://goo.gl/naZX8S

Examples	Security Object	StanfordNLP (sentence)	Ours (object)
Please block the location	Location	Positive	Negative
Never disable my location	Location	Negative	Positive
Do not share my location	Location	Negative	Negative

Table 1: Difference between sentimental analysis towards a sentence and a security object/word. The example shows inconsistent sentimental values between sentences and security objects. Comparing with stanfordNLP, we use sentimental analysis for a different purpose. We use the analysis for generating rewriting specifications. The positive/negative in stanfordNLP means the overall combination of positive and negative words in a sentence. The positive/negative in *IronDroid* means the trustworthiness towards a security object from a user perspective. In the first example, StanfordNLP identifies the sentimental value is positive because the overall attitude of this sentence is positive. *IronDroid* identifies the sentimental value for location is negative because the user does not want to share location.

data usage inside an app. Private data can be potentially leaked via a sensitive sink without user notification.

Challenges. Identifying semantic elements and dependence relations in sentences has been studied in the NLP community. However, these techniques like name entity recognition [27] or relation extraction [7] *cannot* be directly applied for our purpose. They are not designed for the domain of app customization. Furthermore, app rewriting modifies programs on the code level. Programs consist of type-formatted code, which expresses little semantic information. Developing such tool needs comprehensive considerations across different domains. The main technique challenge is the mapping natural languages to program inputs for rewriting. It would be very difficult in practice to process users sentences in free forms. In our design, we require the sentences in a structured format that can be parsed by our prototype. We discuss how to deal with more natural expressions as our future direction in Section 6. In the example, the user concerns on the device information, and cares whether it is leaked by text messages. We use this sentence as a running example [7] through our analysis.

> Never share my device information, if it is sent out in text messages.

We aim to rewrite an app based on the above sentence. In summary, we face two major technical challenges:

- **Sentimental intention towards security objects**. Given a sentence, we extract both the security-related object and the sentimental value towards the object. Table 1 demonstrates different goals for traditional NLP analysis and ours. Traditional NLP analysis identifies the sentimental value for the whole sentence [37]. In contrast, our approach detects user intentions toward a security-related object. Extracting user intentions for personalized security is challenging. We build our graph mining techniques to extract security objects. An alternative method is a keyword-based searching algorithm. We demonstrate the imprecision of the keyword-based searching in the evaluation. The imprecision is due to the lack of sentence semantic analysis.

[7]In our prototype, we expect users express the sentence in a structured and clear format.

User input examples	Object p	Source APIs
	Sentimental s	**Rewriting Op**
	Constraint c	**Sink APIs**
Never share my device information, if it is sent out in text messages.	Phone	getDevieId() getSubscribeId()
	s = -1	u.stop | u.trigger
	sent out in text messages	sendTextMessage() sendDataMessage()
Do not release my location outside this app.	Location	getLatitude() getLongitude()
	s = -1	u.stop | u.trigger
	outside this app	*
Do not share my location, if it is sent out to the internet.	Location	getLatitude() getLongitude()
	s = -1	u.stop | u.trigger
	to the internet	HttpExecute() SocketConnect() URLConnect()
The app can share my audio content if needed.	Audio	AudioRead()
	s = 1	-
	if needed	N/A

Table 2: Examples of extracting user intention predicates and mapping them to taint flow analysis inputs. We aim to generate a mapping from a language to code-level inputs. * means we return all possible functions. N/A means we could not find a semantic mapping. In rewriting, *u.stop*|*u.trigger* means the invocation of the unit is prohibited if invoking the unit triggers the privacy violation.

- **Semantic gaps between human languages and app rewriting**. Human languages and app rewriting have different representations. Rewriting specifications are identical to program code structures, while human languages have no direct correlations with the code structures. Mapping language-level user intentions to code-level specifications is challenging. We eliminate the semantic gap by identifying *private data leak vulnerabilities*. We semantically map user intentions into program analysis inputs. A user-intention-guided taint flow analysis is introduced for app rewriting.

2.2 Definitions

We give several key definitions used in our model, including user intention predicate, the taint flow graph and rewriting specification. We define user intention predicate T_s for extracting user intentions towards a security object.

DEFINITION 1. ***User Intention Predicate*** *is a tuple $T_s = \{p, s, c\}$ extracted from a sentence S_{user}, p is a security object from the predefined object set P, where $p \in P$. s is a sentimental value $s \in \{1, -1\}$, where -1 means the negative attitude and 1 means the positive attitude. c is the constraint string for the object p.*

$s = -1$ means users do not trust the app and could limit the use of private data that is related to p.

We define a taint flow graph G for app program analysis. Taint flow graph G statically captures the app behaviors with program analysis. We utilize G for a user-intention-guided taint flow analysis.

DEFINITION 2. ***Taint Flow Graph*** *is a directed graph $G(V, E, S, T)$ with source set $S \subseteq V$ and sink set $T \subseteq V$ and $S \cap T = \emptyset$, where for any flow $f = \{v_0, v_1 ... v_n\}$ in G, $v_0 \in S$ and $v_n \in T$ and $e = \{v_i \rightarrow v_j\} \in E$. The flow f represents the taint-flow path from the source v_0 to the sink v_n, which is denoted as $f = \{v_0 \rightsquigarrow v_n\}$.*

We define rewriting specification R_r for app rewriting. The unit u is a line of code in the intermediate representation

(IR). The *op* captures the rewriting operation towards unit u. The *op* is based on the sentimental value s in T_s. If $s = -1$, our rewriting would terminate the invocation of u if the invocation of u violates users' expectations. If $s = 1$, our rewriting would pass the verification of u, which has no impact on app behaviors. If an app contains multiple units in $R = \cup_{i=1}^{k} R_r^i$, we iteratively rewrite each unit u.

DEFINITION 3. ***Rewriting Specification*** *is a tuple* $R_r = \{u, op\}$ *for rewriting an app* A. u *is a unit/statement in the app and op is an operation to rewrite this unit* u. *In rewriting, we have* $A(R) \Rightarrow A'$, *where* $R = \cup_{i=1}^{k} R_r^i$, R_r^i *is the rewriting specification for unit* u_i *and* A' *is the rewritten application.*

Example and Focus. In our running example, for the user intention predicate T_s, we have $p = \{Phone\}$ because the user considers most on the phone data, $s = -1$ because the user does not want to share the data, and $c = \{if\ it\ is\ sent\ out\ in\ text\ messages\}$. We find one unit u for the rewriting specification R_r, u is the exact line of code that sends out the device information, *op* is the operation to modify u. If u sends out the device information at runtime, the invocation of u is terminated.

Table 2 presents examples of user intention predicates. As explained in Section 2.1, mapping sentences to program inputs is difficult. If expressions can be interpreted in multiple ways, identifying such expressions needs additional constraints and conditions. For example, in the sentence "do not share my location when I am at work", it is non-trivial to generate code for intercepting "I am at work". The condition to verify whether a user is at work is not deterministic. Additional context (e.g., location range) is required. Currently, our approach only handles the structured sentences that can be mapped to sources and sinks for program analysis. We utilize static taint flow analysis to identify rewriting specifications. We present steps on transforming from S_{user} and an original app A to a rewritten app A':

$$S_{user}, A \xrightarrow{\text{NLP}} T_s, A \xrightarrow[\text{Constr.}]{\text{ICFG}} T_s, A, G, \xRightarrow{\text{PA}} R, A \xRightarrow{\text{RE}} A'$$

2.3 Workflow

Figure 1 presents the workflow of our approach. We give an overview of the main operations, including extracting user intention predicate, user-intention-guided taint flow analysis and automatic rewriting.

- **Extract User Intention Predicates**. The goal of this operation is to extract user intention predicate T_s. The input is a natural language sentence S_{user}, and the output is the user intention predicate T_s. We generate two stages of sentence parsing trees. A structure parsing tree is used to analyze the sentence structure. A dependence parsing tree is used to analyze the semantic dependence relations. We utilize graph mining to extract T_s from parsing trees. Section 3 describes the algorithm for user intention extraction.

- **User-intention-guided Taint Flow Analysis**. The goal of this operation is to generate the rewriting specification R_r. The input is the user intention predicate T_s, and the output is the rewriting specification R_r. A static taint flow analysis is used to construct rewriting specifications. Section 4 describes the algorithm for user-intention-guided taint flow analysis.

- **Automatic Rewriting**. The goal of this operation is to automatically rewrite an app A to protect privacy. The input is the rewriting specification R_r, and the output is the rewritten app A'. An app is rewritten based on R_r. The control and data flow integrity are investigated to guarantee the validation of rewriting. Section 5 describes our rewriting strategies.

3. EXTRACT USER INTENTION WITH NATURAL LANGUAGE PROCESSING

The purpose of our NLP analysis is to identify user intention predicates. The output is a user intention predicate T_s in Definition 1. The input is a preprocessed sentence. The preprocesses include the sentence boundary annotating (e.g., handling period), stemming and lemmatization, removing punctuation and stopwords. We also parse Android API and permission documents to identify a predefined security object set P. We categorize security objects by considering both Android permissions and API document descriptions. For example, the "phone" category covers keywords from both READ_PHONE_STATE and READ_CONTACT permission. To filter out noises, we define a set of negative verbs V_n (e.g., block, disable) and positive verbs V_p (e.g., share, allow). The sentimental verb set is associated with user attitudes toward the security object. We filter out the sentence if it does not contain any sentimental verb. We utilize two stages of parsing trees: structure parsing tree T_{parse} for sentence structure analysis and semantic dependence parsing tree T_{depen} for semantic dependence analysis.

Structure Parser. We utilize T_{parse} to identify different structure levels of sentences, e.g., clause level and phrase level. Figure 2 shows how to identify the subordinate clause given a sentence S. The advantage of recognizing the subordinate clause is to increase the accuracy in identify the security object. In a sentence "stop sharing my location if it sent to other devices", the object is the only "location" in the main clause. The "device" is used as the constraint context for the object "location". A keyword-based solution incorrectly identifies both "location" and "device" as objects.

Semantic Dependence Parser. Semantic dependence tree T_{depen} represents semantic relations between words. Figure 3 presents an example of the semantic dependence tree in the main clause. In the graph T_{depen}, each term is followed by a part-of-speech (POS) tag and the type of semantic dependency. We utilize T_{depen} to construct the dependence path $\vec{P}$ from an object p. In Figure 3, the dependence path from object "device" is $\vec{P} = \{share \rightarrow information \rightarrow device\}$.

Word Semantic Similarity. Word semantic similarity is used for calculating the relatedness between a pair of words. Given two words w_1 and w_2, we define the similarity score as $sc = f_{sim}(w_1, w_2)$, where $sc \in [0, 1]$. $f_{sim}()$ is the function to compute similarity score between w_1 and w_2. We identify w_1 and w_2 are a pair of semantic similar words if sc is larger than a threshold. We utilize word semantic similarity to increase the feasibility of object matching in our approach.

3.1 Graph Mining on Parsing Trees

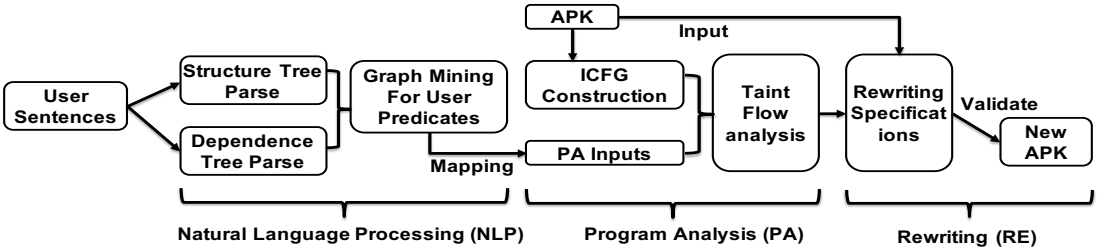


Figure 1: Workflow of our approach. Our approach consists of three major components: NLP is used for transforming natural sentence into user intention predicates, PA is used to perform user-intention-guided taint flow analysis, RE is used to automatically rewrite the app.

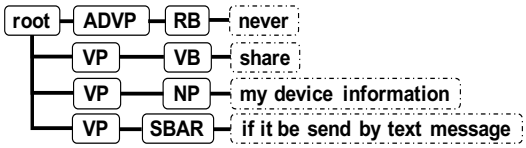


Figure 2: A simplified structure parsing tree annotated with structure tags. The tree is generated from the PCFG parser [22]. The bold rectangle node represents the part-of-speech (POS) tag. The dotted rectangle node represents the word and sentence. The edge represents a hierarchy relation from the root to words. S refers to a simple declarative clause, ADVP refers to adverb phase and VP refers to a verb phrase. The subordinate clause (SBAR) is identified correctly by the parser.

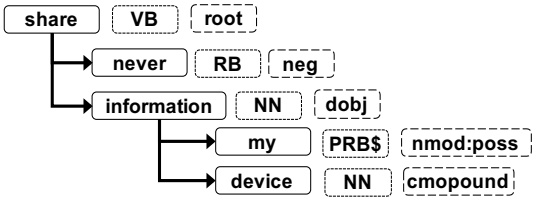


Figure 3: A example on the sentence annotated with semantic dependencies in the main clause. The tree is generated with Stanford-typed dependencies [11]. The arrow represents dependence relations. Each term is followed by two tags. The first tag is the part-of-speech (POS) tag, and the second tag represents the semantic relation (e.g., dobj means the direct object relation).

The purpose of graph mining is to extract user intention predicate T_S. We extract $T_S = \{p, s, c\}$ from two stages of parsing trees.

Security Object Identification. To identify the object, we first identify the main clause S_{main} of the whole sentence S_{whole} by using the parsing tree T_{parse}. The subordinate clause S_{sub} is identified at the same time. We have $S_{main} \cup S_{sub} = S_{whole}$ and $S_{main} \cap S_{sub} = \emptyset$. For cases that users use the prepositional phrase (PP) as the constraint context. We also identify the prepositional phrase S_{pp} inside the main clause, where $S_{pp} \subseteq S_{main}$ and $S_{pp} \cap S_{sub} = \emptyset$. We identify an object p by recursively searching from the main clause to the subordinate clause. In the matching process, we aim to match both a word and its POS tag (NN). POS tags help us efficiently mark a word as noun, verb, adjective, etc. Word semantic similarity increases the tolerance in the matching. In Figure 3, we identify "device" as the object p in the category "phone", because "device" is a noun (NN as the POS tag) and shares a similar semantic meaning with "phone". In another case, we do not detect "contact" as an object in "Siri, contact my mum that I am late for home". The POS tag of "contact" is VB (verb), it is used as an operation rather than an object. Our approach returns null if it cannot detect any object through all the steps.

Sentimental Intention. After identifying p, we construct the dependence path $\vec{P}_p = \{w_i | w_i \in S_{whole}\}$ in T_{depen}. We calculate the sentimental intention through the backward propagation in $\vec{P}_p$. We focus on two negative concepts: negative verbs V_n and negative relations R_n. Negative verbs represent the negative intention towards the object p. Negative relations represent the negative dependencies between two words. The negative relation is marked as "neg" in the dependence graph T_{depen}. The sentimental value s over an object p is computed by the total amount of negative concepts along the path $\vec{P}_p$. If the total amount is odd, we denote $s = -1$. If the amount is even (e.g, double negation), we denote $s = 1$. In Figure 3, the $s = -1$ because "share" and "never" remains a negative relation.

Constraint Detection. Constraint c provides additional restrictions on the object p. We identify the constraint c by extracting words in the subordinate clause or the prepositional phrase. Constraint c has a huge variance in different sentences. A simple sentence "please block my location" has no constraint for object "location". A complex constraint (e.g.,"when I am driving my car") has no directly inference for the taint flow analysis. In our approach, we focus on constraint c that demonstrates a sink function of the object p. For example, c= $\{by\ text\}$ means the function of sending text messages.

4. USER-INTENTION-GUIDED TAINT FLOW ANALYSIS

The purpose of the taint flow analysis is to generate a rewriting specification R_r in Definition 3. The input is the extracted user intention predicate T_s in Definition 1. T_s is semantically mapped to program inputs that can be used for taint analysis. The user-intention-guided taint flow analysis detects sensitive data flows that violate users' expectations. The output of the taint flow analysis is the sensitive sinks with privacy leak vulnerabilities.

Semantic Mapping. The taint flow analysis is guided by the user intention predicate T_s. We map T_s into API inputs, including sources and sinks. For $p \in T_s$, we map p to a set of sensitive sources. For $c \in T_s$, we map c to a set of sensitive sinks. Sources are associated with the security

object p because private data is read from sources. Sinks are associated with the constraint c because private data is sent out by sinks. In the source mapping, we map p by its category into a predefined set of source APIs. In the sink mapping, we parse the sink list from Susi [35] to generate a lookup table $H(K) = T$, where K is a keyword set and T is the sink set. The lookup table maps a constraint c to a set of sinks $\bigcup_i t_i$, where each t_i is a sink function. We have $k \in c$ and $H(k) = \bigcup_i t_i$, where $k \in K$ and $t_i \in T$. Our approach returns all possible sinks if it cannot detect any matched constraint c.

Taint Flow Analysis. We generate rewriting specifications based on the taint flow analysis. The taint flow analysis is used to identify private data leak vulnerabilities. Reachability between a source and a sink is evaluated in the G_{icfg} and the output is a taint flow graph $G(V, E, S, T)$. A taint flow path $f = \{v_0 \rightsquigarrow v_n\}$ represents a potential data leak from v_0 to v_n. We generate rewriting specifications based on a taint flow path f. We identify the unit u in $R_r = \{u, op\}$ as $u = v_n \in f$. The unit u is uniquely identified by matching three types of signatures: the class signature, the method signature and the statement signature. The class signature represents the class of a sink. The method signature represents the method in the class. The statement signature represents the exact line of code in a method.

5. AUTOMATIC REWRITING

The purpose of rewriting is to automatically modify the code based on a rewriting specification R_r. We develop a new rewriting framework to recognize rewriting specifications. Although there exist some rewriting solutions on Smali bytecode (e.g., I-ARM-Droid [13] and RetroSkeleton [12], we choose to reimplement our rewriting framework on Jimple for compatibility and reliability. Jimple is a three-address typed intermediate representation (IR) transformed from Smali by the Soot [23], which allows creation and reorganization of registers. Rewriting on Jimple is less error-prone than the direct modification on Smali. The modified IR is re-compressed as a new APK file and signed with a new certificate. The new APK remains the original logic and functions. Only sensitive sinks are rewritten for privacy protection enforcement.

In this paper, we only demonstrate the use case of rewriting for privacy protection. The rewriting framework can be used for general purposes, e.g., repackaging and vulnerability mitigation. We propose three basic operations in our rewriting framework.

Unit Insertion. We would insert new local variables, new assignments, Android framework APIs and calls to user defined functions into a method. We first need to inspect the method's control flow graph (CFG) and existing variables. We generate new variables for storing parameter contents and new units (e.g., add expression, assignment expression) for invoking additional functions. These new variables and units are inserted before u in R_r.

For Android framework APIs, we generate a new $SootMethod$ with the API signature, and insert it as a callee in the app. We generate an invoking unit as a caller to call the callee in the method. To guarantee the consistence of parameters of the API, we inspect the types of local variables and put them into the caller function in order.

For user defined functions in Java code, we generate a new $SootClass$ with the class and method signature. We insert the $SootClass$ as a new class. We invoke a user defined function in the method by generating a new invocation unit. The Java code is compiled into Jimple IR and attached to the app as a third-party library.

Unit Removal. Unit removal is more straightforward, as we directly remove the unit u in the original function. However, we need to guarantee that the removal unit has no-violation of the control-/data-flow integrity on the rest code. We analyze the CFG of the function to guarantee the validation of removing a unit.

Unit Edition. Unit edition modifies the unit u by changing its parameters, callee names or return values. We decompose u into registers, expressions and parameters, and modify them separately. For example, for a unit as a invocation $u = r_0.sendHttpRerquest(r_1...r_n)$, we decompose it as a register r_0, a function call $sendHttpRerquest$ and the parameters $\{r_1, ...r_n\}$. $sendHttpRerquest$ is replaced with $myOwnHttp$ by implementing a customized checking function. We are able to replace parameter r_0 with a new object r_{new} for monitoring network traffic.

We utilize basic operations to generate complex rewriting strategies. An if-else-condition can be decomposed as a unit insertion (initialize a new variable as the condition value), a unit edition (get condition value) and two unit insertions (if unit and else unit). Users can customize rewriting strategies in rewriting. In our approach, $op \in R_r$ is dependent on the sentimental value $s \in T_s$. If $s = -1$, we define our rewriting strategy is to insert a check function $check()$ before u. The check function terminates the invocation of u if privacy violation happens to u at runtime. This rewriting strategy enforces privacy protection with dynamic interruption.

The complexity of rewriting an app is $O(MN)$, where M is the number of units for rewriting and N is the lines of code. Given a rewriting specification $R_r = \{u, op\}$, the searching algorithm is linear by matching the unit in an app. The time for operation op is constant.

6. LIMITATION AND DISCUSSION

In our design, we mainly focus on the technical challenges for developing a natural language interface to programs for rewriting. User interfaces and legal issues (e.g., copyright restrictions) are out of the scope of discussion.

NLP Limitation. Our study shows higher precision in understanding user intentions than the state-of-the-art approach for app rewriting. In our prototype, we expect users express the sentence in a structured format that can be parsed by our prototype. However, our approach still introduces some false positives and incorrect inferences. The inaccuracy most comes from abnormal ways of user presentations, e.g., ambiguous grammars and complex expressions. Also, the standard dependence parsing cannot correctly infer the semantic relations for complex sentences sometimes. The inaccuracy of dependence parsing further introduce wrong identifications of security objects and the sentimental value. In the future work, more efforts are required to better resolve above limitations.

Taint Analysis Limitation. Our prototype is built on the static program analysis framework [6]. The inaccuracy of our prototype comes from the inherent limitations of the static analysis. Static analysis cannot handle dynamic code obfuscation. Static analysis often over-approximates taint flow paths. The detected taint flow paths may not be feasible at runtime. We mitigate limitations of static analysis

with an efficient rewriting mechanism. Our rewriting framework can terminate a function's invocation only when privacy violation happens. We plan to extend our prototype with a more capable hybrid analysis. The hybrid analysis enhances the resistance of dynamic obfuscations.

Rewriting Limitation. Our rewriting framework provides multiple operations for app customization. The research prototype is based on the Soot infrastructure with the Jimple intermediate representation. The Jimple IR is extracted from Android Dex bytecode by app decomposition. We recompile Jimple IR into Dex bytecode after rewriting. ART and Dalvik virtual machines [8] are compatible to run Dex bytecode. The efficiency of rewriting relies on app decomposition. Current decomposition frameworks [30, 16] cannot handle native code and dynamic loaded code. Similar to static analysis, we cannot rewrite bytecode that is not compatible to Jimple. How to detect dynamically loaded code is studied recently in [33]. In the future work, we plan to enhance rewiring by increasing decomposition capacity.

Deployment Limitation. Our research goal is building a generic tool to support flexible natural language inputs. Everyone could use the tool as an interface to specify security policies and customize apps. However, the usability of our prototype is limited to current techniques, e.g., the performance bottleneck of taint flow analysis. The mitigation of limitations can be resolved by choosing a low-precision taint flow analysis configuration, e.g., context/flow-insensitive module. The prototype only demonstrates a possible language interface to programs for app rewriting. More efforts are needed to improve the precision in both language understanding and program analysis.

7. EXPERIMENTAL EVALUATION

We present the evaluation of our approach in the section. We list four research questions in our evaluation. **RQ1:** What is the accuracy of *IronDroid* in identifying user intentions? (precision, recall and accuracy) **RQ2:** How flexible is *IronDroid* in rewriting apps? (success rate and confirmation of modified behaviors) **RQ3:** What is the performance overhead of *IronDroid* ? (size overhead and time overhead) **RQ4:** How to apply *IronDroid* for practical security applications? (security applications)

To answer the research questions, we evaluate the accuracy of user intention extraction in Section 7.2. We evaluate the feasibility of rewriting in Section 7.3. We measure the performance of our approach in Section 7.4 We provide three case studies in Section 7.5.

7.1 Experiment Setup

We implement our research prototype by extending StanfordNLP library for user intention analysis. We utilize PCFG parsing [22] for structure parsing and Stanford-Typed dependencies [11] to detect semantic relations. We use WordNet [32] similarity to compute the relatedness between two words. Our taint flow analysis is based on the cutting-edge program analysis tool FlowDroid [6]. We extend FlowDroid to support natural language transformation and rewriting specification generation. We implement our own Android rewriting framework with Jimple IR based on Soot.

To evaluate the accuracy of user intention analysis, we evaluate totally 153 sentences collected from four users with different security backgrounds. Users express their concerns on the personal information. The average length of a sentence is 10. 30.1% of sentences contain subordinate clauses. We have two volunteers to independently annotate each sentence in the corpus. Each sentence is discussed to a consensus. We use the corpus as ground truth for user intention analysis. For the taint flow analysis and app rewriting, we evaluate benchmark apps (total 118 apps) from DroidBench. We dynamically trigger sensitive paths in rewritten apps for validation. We compute time and size overhead for the performance evaluation. In security applications, we show how to potentially apply our approach to privacy protection. We also demonstrate how to achieve a more fine-grained location restriction. The location restriction is beyond the current Android dynamic permission mechanism.

[8]https://goo.gl/EvdMdO

7.2 RQ1: User Intention Accuracy

The output predicates $T_s = \{p, s, c\}$ is a set of tuples. We mostly focus on the accuracy of the user object p and the user sentimental value s. These two concepts are most important to express user concerns and attitudes. The constraint c has a huge variance and is hard to perform standard statistics evaluation. We compare precision with following models:

- **StandfordNLP** includes a basic sentiment model for sentence-based sentimental analysis. In our problem, StanfordNLP does not provide the security object identification.

- **Keyword-based Searching** identifies the security object and the sentimental value by keyword matching in the sentence. Keyword-based searching ignores the structure and semantic dependencies. Keyword-based searching is regarded as the state-of-the-art analysis in user intention identification.

- ***IronDroid*** captures the insight of sentence structure and semantic dependence relations. *IronDroid* identifies the object and the sentimental value via graph mining from two stages of parsing trees. *IronDroid* also utilizes the semantic similarity for word comparison.

For a scientific comparison, the keyword-based searching and *IronDroid* share the same predefined object set. We identify categories of objects: {*phone, calendar, audio, location*}. The four categories present a high coverage of private data for mobile app users. We measure the precision, recall and accuracy of user sentimental value identification. TP(true positive), FP(false positive), TN(true negative), and FN(false negative) are defined as:

1. TP: the approach correctly identifies the user sentimental value as negative ("not sharing").

2. FP: the approach incorrectly identifies the user intention value as negative ("not sharing").

3. TN: the approach correctly identifies the user intention value as positive ("sharing").

4. FN: the approach incorrectly identifies the user intention value as positive ("sharing").

We further define precision (P), recall (R) and accuracy (Acc) as:

$$P = \frac{TP}{TP+FP}, R = \frac{TP}{TP+FN}, Acc = \frac{TP+TN}{TP+FP+TN+FN} \quad (1)$$

IronDroid achieves the highest accuracy than the other two approaches. Table 3 presents the sentimental analysis of three different approaches, our approach achieves 95.4% accuracy in identifying user sentimental value. StandfordNLP achieves very low accuracy 68.6% in the experiment. We believe the standard NLP model is not suitable for our problem. In our problem, we compute the sentimental value towards the security object. StandfordNLP computes the sentimental value for the whole sentence. There is no direct correlation between two different sentimental levels. For example, standfordNLP identifies "I do care, please block the location information" and "it is okay to share my location information" with the same optimistic/positive attitude. However, the first sentence presents a negative intention (blocking) for the object "location". Therefore, we cannot directly apply standfordNLP for our problem. Keyword-based searching fails because of the lack of insight in sentence structure and semantic dependence. For example, "do not share my location if I am not at home" is misclassified by the keyword-based searching. The main reason is that "not" in the subordinate clause has not direct semantic dependence on the main clause. Our approach has more insights in capturing sentence structures and semantic dependence relations. In summary, our approach is very accurate in understanding user intentions.

Table 4 presents the improvement of *IronDroid* comparing with the keyword-based searching for the object identification. For each category, we define the improvement in precision, recall and accuracy as:

$$\begin{aligned} \triangle P &= P_{Ucer} - P_{keyword} \\ \triangle R &= R_{Ucer} - R_{keyword} \\ \triangle Acc &= Acc_{Ucer} - Acc_{keyword} \end{aligned} \quad (2)$$

How to compute the precision, recall and accuracy for each object can be easily referred as the user sentimental evaluation. Our results show that, comparing with keyword-based searching, *IronDroid* effectively identifies objects with the average increase in precision, recall and accuracy of 19.66%, 28.23% and 8.66%. The improvement comes from two major reasons: word semantic similarity to increase true positives and dependence analysis to reduce false positives. Keyword-based searching introduces large false positives by over-approximating security objects. For example, in the sentence "I do not want share my calendar information if it is sent to other devices", keyword-based searching identifies both "calendar" and "device" as security objects because of the lack of structure analysis.

Table 5 presents the overall accuracy for user intention identification. The evaluation is based on the combination of the sentimental intention s and the object p. We mark the result $\{p, s\}$ true only when both s and p is identified correctly. *IronDroid* achieves 90.2% accuracy rate, while keyword-based approach only achieves 53.6% accuracy rate. Our approach has nearly 2-fold improvement than the state-of-the-art approach. The experimental results validate that our approach is effective in identifying user intentions from natural languages.

	TP	**FP**	**TN**	**FN**	**P**(%)	**R**(%)	**Acc**(%)
StandfordNLP	55	18	50	30	75.3	64.7	68.6
Keyword-based	64	9	63	17	87.7	79.0	83.0
Ours	68	5	78	2	**93.2**	**97.1**	**95.4**

Table 3: Sentimental analysis accuracy for *IronDroid* , StanfordNLP and keyword-based searching. *IronDroid* achieves higher accuracy, precision and recall than the other two approaches. our approach effectively achieves the improvement in 19.66% for precision, 28.23% for recall and 8.66% for accuracy on average.

Object	**△P** %	**△R**%	**△Acc**%
Location	19.23	14.52	11.11
Phone	20.00	38.15	9.80
Calendar	26.09	28.57	7.84
Audio	13.33	31.66	5.88
Average	**19.66**	**28.23**	**8.66**

Table 4: object detection improvement comparing with keyword-based approach. Our approaches have the average increase in precision, recall and accuracy of 19.66%, 28.23% and 8.66% for four categories of objects.

	Keyword-based	***IronDroid***
Accuracy	53.6%	**90.2**%

Table 5: The overall accuracy for understanding sentimental and the object. Our approach has a significant improvement than the keyword based searching approach.

7.3 RQ2: Rewriting Robustness

DroidBench is a standard test suite for evaluating the effectiveness of Android program analysis [20, 6]. DroidBench is specifically designed for Android apps. We tested our rewriting capability based on the benchmark apps. We use the example "never share my device information, if it is sent out in text messages" as the input for user intention analysis. We generate the user intention predicate T_s as $p =$ "Phone", $s = -1$ and $c =$"sent out in text messages". In the user-intention-guided taint flow analysis, we map T_s to a list of code inputs. Table 6 presents how we map "Phone" into source APIs and "text" into sink APIs. We perform the taint flow analysis in 13 different categories among 118 apps. Each category represents one particular evasion technique (e.g., callback). Without loss of generality, we randomly choose an app for rewriting in each category. The tested apps have a high coverage to mimic real-world apps. We identify a taint flow from a source to a sink within the taint flow graph. We generate the rewriting specification R_r by extract the unit u of the sink. For the rewriting operation op, we use unit removal to remove the sensitive sink API. We use unit insertion to add a dynamic logging function. The sink API is the leaf node in the ICFG, the unit removal has no impact on the rest code. To evaluate the feasibility of the rewriting, we install all the rewritten apps in a real-world device Nexus 6P with Android 6.0.1. We aim to answer two major questions:

1. Whether these apps remain valid program logics (do not break the functionality)?
2. Whether we would observe the modified bahavior at runtime (protect privacy)?

To answer the first question, we install rewritten apps on the device. We utilize *monkey* tool to generate a pseudo-

Category	Mapping
Device info	getDeviceId(),getSubscriberId(), getSimSerialNumber(), getLine1Number()
Sent by text	sendTextMessage(),sendDataMessage(), sendMultipartTextMessage(),sendMessage()
Never share	unit removal (sink unit), unit insertion (log unit)

Table 6: Mapping the sentence to analysis inputs for rewriting in the experiment. The sentence is from the example "Never share my device information, if it is sent out in text messages".

Category	AppName	PA time	RE time	Run	Logging Confirm
Alias	Merge1	1.74	6.40	✓	✓
ArrayList	Array Access1	1.13	4.34	✓	✓
Callbacks	Button1	2.52	4.48	✓	✓
FieldandObj ctSensitivity	Inherited Objects1	1.27	4.24	✓	✓
InterAppCom	N/A	-	-	-	-
InterCom ponentCom	Activity Communi cation1	1.08	4.51	✓	✓
LifecyCycle	ActivityLife cycle2	1.15	4.20	✓	✓
GneralJava	Loop2	1.14	4.18	✓	✓
Android Specific	DirectLeak1	1.11	4.10	✓	✓
ImplicitFlows	N/A	-	-	-	-
Reflection	Reflection1	1.22	4.27	✓	✓
Threading	N/A	-	-	-	-
Emulator Detect	PlayStore1	1.90	6.46	✓	✓
Summary	10			**10**	**10**

Table 7: Runtime scalability for testing benchmark apps. N/A means we did not find any app matched the user command with in the taint analysis. Run w. monkey means the rewritten apps can run on the real-world device Nexus 6P. We use *monkey* to generate 500 random inputs in one testing. Confirmation means we detect the modified behavior using the adb *logcat*. PA is short for taint-flow-based program analysis time, RE is short for automatic rewriting time.

random stream of user events. We generate 500 random inputs in testing one app. We perform the stress test on the rewritten app. To answer the second question, we need to trigger taint flow paths and observe the modified behaviors. We use *logcat* to record the app running state. The dynamic logging function $Log.e()$ can be captured by *logcat* at runtime. To increase the coverage of user interactions, We also manually interact with the app, e.g., clicking a button. If we observe such log information, the behavior of an app is modified and we validate our rewriting on the app.

Table 7 presents the robustness of our rewriting. Our approach successfully runs and detects all the rewritten apps. For example, we detect a taint flow path as $f = \{getDeviceId() \rightsquigarrow sendTextMessage()\}$ in $Button1.apk$. Our user-intention-guided taint flow analysis successfully captures the sensitive taint path triggered by a callback $onClick()$. Our rewriting technique identifies the taint flow path f and the rewriting unit $sendTextMessage()$. We modify the code in the app without violates its control and data dependence. These steps guarantee the success of our rewriting technique. In summary, our approach is very efficient in rewriting apps.

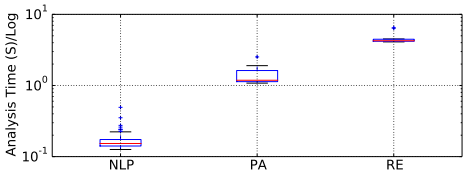


Figure 4: Time overhead for analysis, the average time for NLP analysis is 0.16 second, the average time for PA analysis is 1.43 seconds, the average for RE is 4.68 seconds.

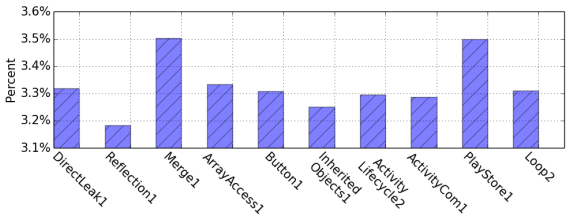


Figure 5: Size Overhead for benchmark apps, our rewriting introduces 3.3% percent overhead on file size for benchmark apps.

7.4 RQ3: Performance Overhead

We compare the runtime overhead of natural language processing (NLP), taint flow analysis (PA) and rewriting (RE) in Figure 4. Experiments were performed over on a Linux machine with Intel Xeon CPU (@3.50GHz) and 16G memory. Figure 4 presents the three runtime distributions in log scale. The average time for NLP is 0.16 second. The average time for PA is 1.43 seconds. The average time for RE is 4.68 seconds. The average runtime of RE is larger than that of NLP and PA. RE needs to decompile each class for inspection and recompile the app after rewriting.

Figure 8 presents the size overhead of rewriting benchmark apps. The size overhead comes from two major sources: *1)* the complexity of rewriting specifications; *2)* the number of impacted code in rewriting. In the experiment, the average file size of the benchmark apps is 305.23 KB. Our approach achieves 3.3% size overhead on average, which is relatively negotiable. In summary, our approach is very practical in rewriting apps.

7.5 RQ4: Security Applications

In this section, we show how to extend *IronDroid* for real-world problems. We present three security applications that cover *1)* proactive privacy protection, *2)* mitigating ICC vulnerability and *3)* location usage restriction. These security applications demonstrate the feasibility of our rewriting framework. Table 8 presents the statistics of three demo apps.

7.5.1 Proactive Privacy Protection

The purpose of this demo is to show the rewriting flexibility for normal users. *com.jb.azsingle.dcejec* is an e-electronic book app. The app passes all the 56 anti-virus tools for vetting screening. The user concerns on the data leakage to the storages. Users can proactively restrict the data usage. The sentence is that "do not reveal my phone information to outside storages". We use this example to show how a user could use *IronDroid* for a proactive privacy protection. Based on the our NLP analysis, we identify user in-

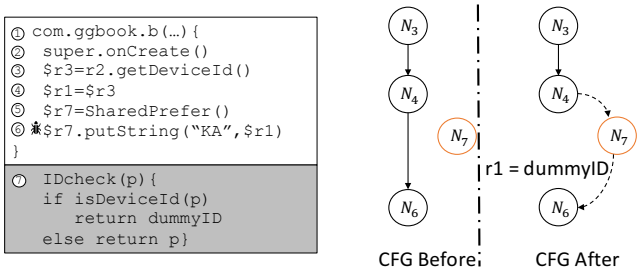
Package	Report	Label
com.jb.azsingle.dcejec	0/56	benign
com.xxx.explicit1	1/57*	grayware
com.bdsmartapp.prayerbd	0/56	benign

Table 8: All three apps pass the vetting screening of antivirus tools. * means the alert mentions it contains a critical permission *READ_PHONE_STATE* without any additional information. We identify the second app as grayware [4]. Users have the preference for customizing these apps for personalized security.

tention predicate T_s = {"phone", -1, "outside storage"}. The user-intention-guided taint flow analysis identifies a critical taint flow graph f_1 in a class *com.ggbook.i.c*. The taint flow represents that the phone information is written into a file *KA.xml*. The sink unit is a sensitive function *putString*(). *KA.xml* is located in the device storage. The taint flow path f_1 is presented as below:

- f_1: [r3 = invoke r2.<TelephonyManager: String getDeviceId()>(), r4 = r3, return r4, <com.ggbook.c: String z> = r1, r1 = <com.ggbook.c: String z>, r7 = invoke r7.<SharedPreferences: putString()>("KA", r1)]

Our rewriting is based on the unit that invokes the *putString*() function. The unit is identified as r7 = invoke r7.<SharedPreferences: putString()>("KA", r1). In rewriting, we implement a dynamic checking function *IDcheck*(). *IDcheck*() accepts the parameter $r1$ from the unit. The type of $r1$ is Java.lang.String. *IDcheck*() dynamically checks $r1$ by comparing the value with the actual device ID p = *getDeviceId*(). If p = *getDeviceId*() $\in r1$, *IDcheck*() returns a dummy ID and overwrite the parameter $r1$. The dummy ID can be further used without violating data flow dependencies. Figure 6 presents the code snippet and the CFG after we insert the *IDcheck*() function.



(a) A simplified code structure of the method. The IDcheck function is inserted after rewriting.

(b) The before and after CFG. The CFG is changed. The privacy is protected by rewriting.

Figure 6: The demo of code snippet and the modified control flow graph (CFG). We insert a IDcheck() function as a new node in the original CFG.

To validate our rewriting feasibility and dummy ID insertion. We run the app with *monkey* on a real device. We manually find the shared preference file in the app file directory. The shared preference file is stored in /data/data/com.jb.azsingle.dcejec/shared_prefs, we identify the dummy ID as 12345678910. The entry *IsImei* = *true* suggests that the dummy ID is successfully inserted. The private phone data is protected from outside storages.

```
<map><string name="KA">12345678910</string>
<boolean name="IsImei" value="true"/></map>
```

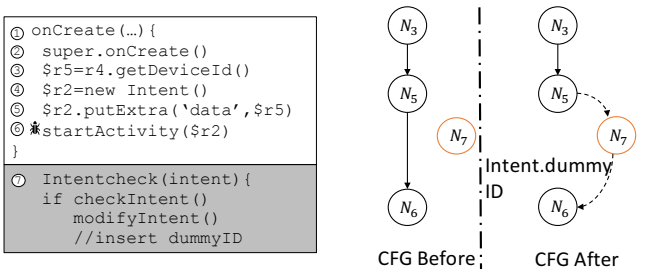
7.5.2 *Mitigating ICC Vulnerability*

The purpose of this demo is to present the rewriting feasibility on mitigating vulnerabilities. In this example, we focus on preventing inter-component communication (ICC) vulnerability. Existing detection solutions aim at detecting vulnerable ICC paths [14, 29] with static program analysis. How to avoid realistic ICC vulnerabilities at runtime is still an open question.

We apply *IronDroid* to showcase a practical mitigation solution to prevent ICC vulnerability. *com.xxx.explicit*1 is an app in ICCBench[9]. The app contains a typical ICC vulnerability. The sensitive phone data (IMEI) is read from a component. The IMEI is put into a data field in an intent. The data is sent out to another component by the intent. The receiver component has access to IMEI without acquiring for the permission READ_PHONE_STATE. The ICC vulnerability results in a privilege escalation for the receiver component.

Our rewriting is based on a user command: "do not share my device information with others". The user-intention-guided taint flow analysis identifies a critical taint flow as f_2 in a class *explicit*1.*MainActivity*. The taint flow path f_2 is presented as:

- f_2: [r5 = invoke r4.< TelephonyManager: getDeviceId() > (), invoke r2.< putExtra(String,String)>("data",r5), invoke r0.< startActivity(android.content.Intent) >(r2)]



(a) A simplified code structure of the method. The IDcheck function is inserted after rewriting.

(b) The before and after CFG. The CFG is changed and the vulnerability is mitigated by rewriting.

Figure 7: The demo of code snippet and the modified control flow graph (CFG). We insert a Intentcheck() function as a new node in the original CFG.

Our rewriting specification focuses on the unit that invokes the *startActivity* function. *startActivity* is used for triggering an ICC. The intent $r2$ is an object that consists of the destination target (e.g., the package name) and data information (e.g., extras). Replacing intent would cause runtime exceptions. The communication of components is based on intents. The component that can receive the intent is defined in the package field of an intent. We aim to protect user privacy without interrupting communication channels. In rewriting, we implement a hierarchy checking function *Intentcheck*(). *Intentcheck*() accepts the parameter of an intent and modify the intent in-place. *Intentcheck* can recursively check all the fields in an intent and rewrite the intent. In this example, we rewrite an intent by inserting dummy data in the data field. Figure 7 presents the code snippet and the CFG after we insert the *Intentcheck*() function.

[9]https://goo.gl/EPjb3n

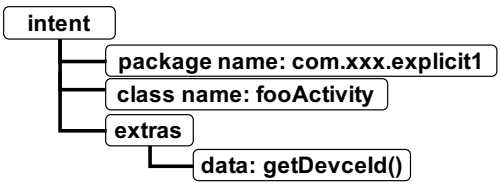


Figure 8: The hierarchy of an intent object. The privacy data getDdeviceId() is stored with a key *data* in a HashMap in *extras* filed.

To validate the modified intent, we run the app with *monkey* on a real device and manually intercept the receiver component. We successfully detect the dummy ID as 12345678910 in the data field of *getIntent*().

The above example demonstrates how a normal user could use *IronDroid* to reduce ICC vulnerability. For better privacy, our rewriting is also capable of redirecting an intent. One possible way to redirect an intent for security is changing an implicit intent to an explicit intent. In the explicit intent, the destination of the intent can be verified by our rewriting. We plan to extend our approach to mitigate other vulnerabilities, e.g., adversarial advertisement libraries.

7.5.3 Location Usage Restriction

The location is a primary concern for most users. *com.bdsmartapp.prayerbd* is an app to provide location-based praying service. It is unclear whether the app would abuse the location and send it to untrusted websites. The purpose of this demo is to show a fine-grained control of location usage. Current Android dynamic permission mechanism only provides turning on and off of the location permission. Our approach achieves extended functionalities: *1) Target inspection*, a user can inspect a targeted function (e.g., to web servers, by text messages and to other apps). *2) Destination inspection*, a user can inspect whether the packet is sent to an untrusted destination. *3) Sensitivity restriction*, a user can protect sensitive locations by sending dummy locations.

We extend our approach to providing a fine-grained restriction with a *LocationCheck* function. We rewrite the app based on the user sentence "do not send my home location to untrusted websites". We extracted user intention predicate T_s ={"Location", -1, "to untrusted website" }. The security object p = {"Location"} is mapped to APIs for reading locations (e.g., *getLatitude*() and *getLongitude*()). The the constraint c = {"to untrusted website" } is mapped to APIs for generating network requests (e.g., *URLConnect*()). In addition, we implement a function *Locationcheck* to determine *1)* the home location and *2)* the trustworthiness of a website. Advertisement websites (e.g., Admob, Flurry) are regarded as untrusted. The home location is defined by a location range. If the URL contains untrusted addresses or the current location is in the location range, the URL is modified by *Locationcheck* for privacy protection.

At runtime, a location request googleapis.com/maps/api/geocode/json?latlng=xxx,xxx&sensor=true is captured by *Locationcheck*. The location request is sent to the Google map server with *googleapis*. *Locationcheck* inspects the latitude and longitude in the location request. *Locationcheck* replaces the original location with a dummy latitude and longitude. A dummy location is shown on the screen layout when we test the rewritten app. In this demonstration, *IronDroid* protects user private location by supporting function-level restriction. More advanced version of our approach can be combined with anomaly data detection solutions.

8. RELATED WORK

NLP for Security and Privacy Compared with NLP applications in other areas (e.g., document analyzing), NLP has only been recently introduced in solving security problems. Prior works utilized NLP to discover online security threats [25] and analyze web privacy policies [44]. In the mobile security, Whype [31] and Autocog [34] used NLP to infer permission-related descriptions. Supor [21], Uipicker [28] and UiRef [9] detected sensitive user inputs from Android app layouts. Lei *et al* [24] generates inputs for program testing from input format specifications. Our work showcases a new application by integrating NLP with app customization. We demonstrate that new NLP techniques are required for personalized security challenges. Our approach requires innovative natural language analysis to understand user intentions for rewriting. We expect to extend our prototype with commercial voice control products (e.g., Siri, Alexa) in the future.

Taint Flow Analysis Researchers proposed taint flow analysis to identify data-flow paths from sources to sinks [10]. CHEX [26] and AndroidLeaks [19] detected data flows to find suspicious apps. DroidSafe [20] used a point-to graph to detect sensitive taint flows. FlowDroid [6] proposed a context-, object- and flow-sensitive static analysis to identify taint flows. Our research prototype extends FlowDroid for a precise taint flow analysis. Unlike these approaches aim to identify malware, we use taint flow analysis for personalized security. Our taint flow analysis is used to generate rewriting specifications for app customization.

Android Rewriting The app-retrofitting demonstration in RetroSkeleton [12] aimed at updating HTTP connections to HTTPS. Aurasium [42] instrumented a low-level library for monitoring functions. Reynaud *et al.* [36] rewrote an app security verification function to discovered in-app billing vulnerabilities. Fratantonio *et al.* [18] enforced secure usage of the Internet permission. The rewriting targets and goals in these approaches are specific. Many rewriting targets can be identified through parsing. Our rewriting approach requires more substantial code modification (e.g., parameter inspection, flow analysis). We demonstrate that more capable rewriting techniques are required for personalized security challenges. Our rewriting framework is general and compatible for complex app customization.

ARTist [8] is a compiler-base rewriting tool to modify Android runtime virtual machine. However, compiler-based rewriting relies on a particular Android version and needs modification of the Android system. Our bytecode rewriting modifies an app's bytecode and is compatible for all the Android versions. Bytecode rewriting is more suitable for our scenario for rewriting apps with a natural language interface. The rewritten can run on any Android devices, without root privilege and customized systems.

Defense of Vulnerabilities Pluto [38] discovered the vulnerabilities in advertisement libraries. TaintDroid [15] adopted dynamic taint analysis to track the potential misuse of sensitive data. Anception [17] proposed a mechanism to defend the privilege escalation by deprivileging a portion of the kernel with sensitive operations. DeepDroid [40] applied dynamic memory instrumentations to enforce security poli-

cies. Instead of modifying the kernel or Android framework, our approach focuses on the app-level rewriting. We demonstrate promising security applications of app customization beyond defending vulnerabilities.

9. CONCLUSIONS AND FUTURE WORK

We investigated the problem of app customization for personalized security. We proposed a user-centric app customization framework with natural language processing and rewriting. Our preliminary experimental results show that our prototype is very accurate in understanding user intentions. The prototype is also very efficient in rewriting. We show its applications in providing more fine-grained location control and mitigating existing vulnerabilities. Our approach makes impressive progress toward personalized app customization. More efforts are needed to improve the usability of personalized app customization. For future work, we plan to improve the usability of our prototype with more engineering efforts.

10. REFERENCES

[1] Amazon Alexa. https://developer.amazon.com/alexa.

[2] Android dynamic permission. https://developer.android.com/training/permissions/requesting.html.

[3] Apple Siri. http://www.apple.com/ios/siri/.

[4] Andow, B., Nadkarni, A., Bassett, B., Enck, W., and Xie, T. A study of grayware on Google Play. In *Proc. of MoST* (2016).

[5] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proc. of NDSS* (2014).

[6] 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 *Proc. of PLDI* (2014).

[7] Bach, N., and Badaskar, S. A review of relation extraction. In *Literature review for Language and Statistics II* (2007).

[8] Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., and Weisgerber, S. ARTist: The Android runtime instrumentation and security toolkit.

[9] Benjamin Andow, Akhil Acharya, D. L. W. E. K. S., and Xie, T. Uiref: Analysis of sensitive user inputs in android applications. In *Proc. of WiSec* (2017).

[10] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards taming privilege-escalation attacks on Android. In *Proc. of NDSS* (2012).

[11] Cer, D. M., De Marneffe, M.-C., Jurafsky, D., and Manning, C. D. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *Proc. of LREC* (2010).

[12] Davis, B., and Chen, H. RetroSkeleton: Retrofitting Android Apps. In *Proc. of MobiSys* (2013).

[13] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A rewriting framework for in-app reference monitors for android applications.

[14] Elish, K. O., Yao, D. D., and Ryder, B. G. On the need of precise inter-app icc classification for detecting Android malware collusions. In *Proc. of IEEE Mobile Security Technologies (MoST)* (2015).

[15] Enck, W., Gilbert, P., Han, S., Tendulkar, V., 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. *ACM Transactions on Computer Systems (TOCS)* (2014).

[16] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of android application security. In *Proc. of USENIX Security* (2011).

[17] Fernandes, E., Aluri, A., Crowell, A., and Prakash, A. Decomposable trust for Android applications. In *Proc. of DSN* (2015).

[18] Fratantonio, Y., Bianchi, A., Robertson, W., Egele, M., Kruegel, C., Kirda, E., Vigna, G., Kharraz, A., Robertson, W., Balzarotti, D., et al. On the security and engineering implications of finer-grained access controls for Android developers and users. In *Proc. of DIMVA* (2015).

[19] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. of TRUST* (2012).

[20] Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N., and Rinard, M. Information-flow analysis of Android applications in DroidSafe. In *Proc. of NDSS* (2015).

[21] Huang, J., Li, Z., Xiao, X., Wu, Z., Lu, K., Zhang, X., and Jiang, G. SUPOR: Precise and scalable sensitive user input detection for android apps. In *proc. of USENIX Security* (2015).

[22] Klein, D., and Manning, C. D. Accurate unlexicalized parsing. In *Proc. of ACL* (2003).

[23] Lam, P., Bodden, E., Lhoták, O., and Hendren, L. The soot framework for java program analysis: a retrospective. In *Proc. of CETUS* (2011).

[24] Lei, T., Long, F., Barzilay, R., and Rinard, M. C. From natural language specifications to program input parsers. In *Proc. of ACL* (2013).

[25] Liao, X., Yuan, K., Wang, X., Li, Z., Xing, L., and Beyah, R. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proc, of CCS* (2016).

[26] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of CCS* (2012).

[27] Nadeau, D., and Sekine, S. A survey of named entity recognition and classification. *Lingvisticae Investigationes* (2007).

[28] Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., and Wang, X. Uipicker: User-input privacy identification in mobile applications. In *Proc. of USENIX Security* (2015).

[29] Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proc. of POPL* (2016).

[30] Octeau, D., Jha, S., and McDaniel, P. Retargeting android applications to java bytecode. In *Proc. of FSE* (2012).

[31] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards automating risk assessment of mobile applications. In *Proc. of USENIX Security* (2013).

[32] Pedersen, T., Patwardhan, S., and Michelizzi, J. Wordnet:: Similarity: measuring the relatedness of concepts. In *Demonstration papers at HLT-NAACL* (2004).

[33] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. of NDSS* (2014).

[34] Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., and Chen, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proc. of CCS* (2014).

[35] Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. of NDSS* (2014).

[36] Reynaud, D., Song, D. X., Magrino, T. R., Wu, E. X., and Shin, E. C. R. Freemarket: Shopping for free in Android applications. In *Proc. of NDSS* (2012).

[37] Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., Potts, C., et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proc. of EMNLP* (2013).

[38] Soteris Demetriou, Whitney Merrill, W. Y. A. Z., and Gunter, C. A. Free for all! assessing user data exposure to advertising libraries on Android. In *Proc. of NDSS* (2016).

[39] Tian, K., Yao, D. D., Ryder, B. G., and Tan, G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Proc. of MoST* (2016).

[40] Wang, X., Sun, K., Wang, Y., and Jing, J. DeepDroid: Dynamically enforcing enterprise policy on Android devices. In *Proc. of NDSS* (2015).

[41] Wüchner, T., Ochoa, M., and Pretschner, A. Robust and effective malware detection through quantitative data flow graph metrics. In *Proc. of DIMVA* (2015).

[42] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for Android applications. In *Proc. of USENIX Security* (2012).

[43] Zhang, M., Duan, Y., Yin, H., and Zhao, Z. Semantics-aware Android malware classification using weighted contextual api dependency graphs. In *Proc. of CCS* (2014).

[44] Zimmeck, S., and Bellovin, S. M. Privee: An architecture for automatically analyzing web privacy policies. In *Proc. of USENIX Security* (2014).