# Automated Partitioning of Android Applications for Trusted Execution Environments

Konstantin Rubinov[1], Lucia Rosculete[2], Tulika Mitra[3], Abhik Roychoudhury[3]*
[1]DeepSE Group at DEIB, Politecnico di Milano, Italy
[2]Application Threat Intelligence - Ixia, Romania
[3]School of Computing, National University of Singapore, Singapore
Email: konstantin.rubinov@polimi.it, luciarosculete@gmail.com, {tulika,abhik}@comp.nus.edu.sg

## ABSTRACT

The co-existence of critical and non-critical applications on computing devices, such as mobile phones, is becoming commonplace. The sensitive segments of a critical application should be executed in isolation on Trusted Execution Environments (TEE) so that the associated code and data can be protected from malicious applications. TEE is supported by different technologies and platforms, such as ARM Trustzone, that allow logical separation of "secure" and "normal" worlds.

We develop an approach for automated partitioning of critical Android applications into "client" code to be run in the "normal" world and "TEE commands" encapsulating the handling of confidential data to be run in the "secure" world. We also reduce the overhead due to transitions between the two worlds by choosing appropriate granularity for the TEE commands. The advantage of our proposed solution is evidenced by efficient partitioning of real-world applications.

## 1. INTRODUCTION

Mobile devices have seen the emergence of services that require an increased level of security (e.g., online banking, premium content access, enterprise networks connection). At the same time, these devices adopt open software platforms allowing consumers to install arbitrary third-party applications capable of compromising the critical services. Attackers have responded by investing in exploiting the growing number of such vulnerabilities to gain access to valuable data [10].

As a countermeasure, device manufacturers take a scalable approach to security through hardware protection measures factored in early in the design process. For example, ARM TrustZone [16] is designed to support Trusted Execution Environment (TEE) [39]: a small secure kernel that shares the processor with a rich OS (e.g., Android). TEE provides hardware-enforced security to authorized software (Trusted Applications) protecting them from malware in the rich OS.

Even though TEE provides security guarantees against strong attacks, few applications employ this technology. Enhancing existing mobile applications with TEE requires identification of confidential data and all the operations associated with this data in complex Android applications. Even for in-house applications, this task can prove daunting in the context of Java deep class fields and aliasing.

We propose to fortify existing Android applications by automatically partitioning them with respect to their data protection needs. The partitioning abstracts Java code segments that manipulate confidential data into candidate Trusted Applications to be executed in the secure world with higher privileges, isolated from less privileged malicious or hijacked applications running in the rich OS.

Our approach automatically analyses, partitions, and transforms existing applications for deployment of their confidential data and computations on TEE as Trusted Applications. It automates the laborious and prone-to-human-error tasks such as detection of application paths that propagate and operate on confidential data, application partitioning in the presence of control-dependent secure information flow, extraction and grouping of confidential fragments of application, and interfacing between the normal and secure worlds.

The approach is suitable for complex real-life applications with extended inter-procedural inter-class secure information flow. The analysis is precise; it extends context- and field-sensitive taint analysis of FlowDroid [25]. The refactored app preserves application semantics, satisfies an unidirectional TEE execution model, and maintains a minimal Trusted Computing Base (TCB)[1] after the addition of Trusted Applications. Field-sensitive analysis produces safe data handles (opaque pointers) to enforce information hiding while allowing the rich OS to reference confidential data and thereby reduce the unnecessary communication between the two worlds.

Our automated partitioning framework takes in an application binary as input and outputs the source code of the refactored application with privileged instructions isolated as static Java methods. Presently, we require the developer to manually convert these Java methods to Trusted Applications in native code. But our approach facilitates deployment of Trusted Applications through automatic generation of auxiliary TEE-specific code for establishing interaction between normal and secure worlds. In the future, we plan to take advantage of available Java to C language translators for transformation of Java methods to native code.

Our solution guarantees data integrity, i.e., data cannot be modified in an unauthorized or undetected manner. Con-

*Most of this work was done while the first two authors were affiliated to National University of Singapore.



*ICSE '16, May 14-22, 2016, Austin, TX, USA*
 ISBN 978-1-4503-3900-1…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884817

[1]TCB: a set of components to be trusted to trust a system.

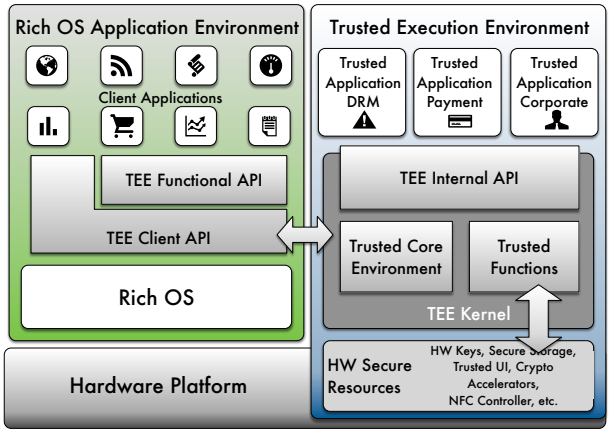


**Figure 1: TEE system architecture**

fidential data and computation on these data reside only in TEE; confidential data can reside in rich OS only in encrypted form. Any data accessed through peripherals under control of TEE is confidential, while any data returned to rich OS is not (as it can be leaked to unauthorized users). The approach allows for controlled release of declassified information. Execution of Trusted Applications in TEE may affect execution of applications in rich OS; however, execution in rich OS cannot interfere with the execution in TEE.

We evaluate our approach with real-world Android applications and micro-benchmarks. Evaluation results highlight the effectiveness of our approach in automatically partitioning complex applications and variety of flow situations arising in Android applications. Trusted Applications maintain a small TCB size of approximately 200–300 lines of code (LOC) and low communication overhead. Overall, the approach enables confidentiality and integrity enforcement through code transformation while substantially reducing development efforts.

## 2. BACKGROUND

Our approach facilitates application development and transformation for Trusted Execution Environment built using ARM TrustZone. ARM TrustZone technology offers a system-wide security solution, partitioning the hardware and software resources so that they exist in one of two worlds: *Secure world* for the security subsystem and *Normal world* for everything else [16]. This is achieved with a set of protective measures integrated in the ARM processor core, memory management unit and the AMBA AXI bus.

Despite wide availability of proprietary implementations of TEE for platforms with ARM TrustZone hardware extensions, there are few Android applications that leverage this technology due to the lack of standardization. This issue is addressed by Global Platform [3] that developed the standard for managing applications on secure chip technology and a set of specifications for the TEE system architecture [11–13]. The overall TEE system architecture is shown in Figure 1.

TEE offers safe execution of authorized software known as *Trusted Applications* (TAs). A TA is composed of *TEE Commands* that collectively provide *secure services* to the clients of the TA, while enforcing confidentiality, integrity and access rights to the resources and data. Each TA is independent and protected against unauthorized access from other TAs, allowing an ecosystem of application providers. TAs can access security resources and services such as key management, cryptography, secure storage, secure clock, trusted display and trusted virtual keyboard via the *TEE Internal API*. Client applications running in the rich OS can access and exchange data with TAs via *TEE Client API*.

Our main contribution is to automatically extract application segments to be deployed as TEE Commands in TA. We automatically generate the auxiliary code to invoke both TEE Client API from rich OS and TEE Internal API from TAs to provide secure services and establish connection between the trusted and un-trusted application segments.

The deployment of TAs requires careful analysis of the capabilities and the constraints inherent to the TEE. A critical constraint imposed by TEE is that TA contributes to the TCB size, which should be kept as small as possible to reduce security risks [28]. Thus TA code should be minimized.

Another constraint is imposed by the unidirectional communication between client and TAs. Rich OS initiates the communication whereby the processor switches to the secure world, executes TEE Command, and finally switches back to the normal world. TAs cannot perform callbacks to the client code. This constraint is especially challenging in the face of complex control flow, and OS- and UI-related operations in Android.

## 3. OVERVIEW

An overview of our approach is shown in Figure 2. The approach builds on several insights that enable extraction of minimal code fragments to TEE and semantic-preserving transformation of the application code.

We highlight the intuition behind our approach using a real-life example – an open-source application *Google Authenticator* [4]. The app supports One-Time Password (OTP) security tokens for two-factor authentication and implements the HMAC-Based (HOTP) and the Time-based (TOTP) OTP generation algorithms. *Google Authenticator* uses either QR code scanning or manual input to obtain an encoded security key issued by Google. The key is decoded and used to calculate a message authentication code (MAC) of a timestamp or a counter to generate OTP. The confidential data in the app includes: security key; user account information; and program state including program clock and an internal counter. For our example, the engineer would first want to protect the confidentiality of the security key by protecting entry points of the key, its manipulation and storage.

Manual inspection of the example revealed that the sensitive data and operations are distributed in six medium to large classes of the application. These are highly UI-dependent classes with complex control flows owing to the event-based nature of the app. There are more than one hundred methods and over 1.6K lines of code (LOC) involved in the propagation of the security key across the classes. Analysis and protection of code fragments of this size and complexity require a systematic approach.

The situation is exacerbated by the presence of overlapping flows of sensitive data from different contexts – a security key coming from a QR code scanning and from manual key entry. The multiple-entry-point components of an Android app make it difficult to establish whether extraction of a specific code fragment affects different application flows. Manual transformation of such an app to deploy sensitive operations and data on TEE is not trivial.

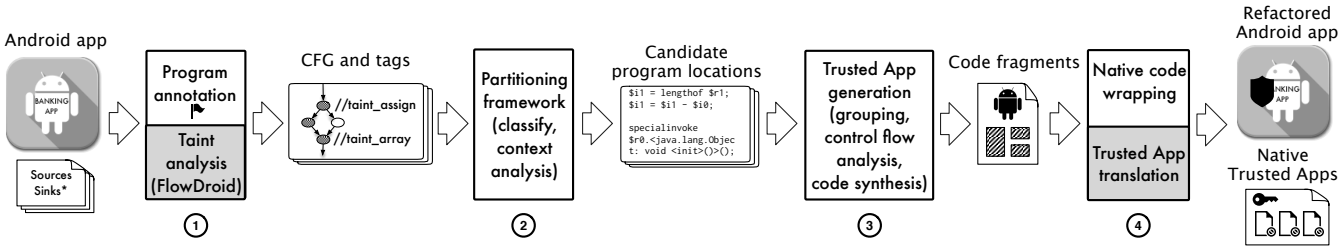


**Figure 2: An overview of the approach**

Our approach does not require the engineer to inspect the complete application to identify critical or non-critical segments thereby relieving him/her of the laborious and error-prone task. The approach starts with an automated context-sensitive taint analysis to track propagation of the confidential data in the program (phase ① in Figure 2). It collects the intermediate analysis data and results, and feeds them to our automated partitioning framework that generates candidate code segments to be deployed as TEE Commands of a Trusted Application, and refactors original app integrating in it TEE Command invocations (phases ②, ③ and ④). Gray areas in Figure 2 highlight the external (in phase ①) or manually supported components (in phase ④).

The approach builds upon and extends the state-of-the-art static taint analysis of FlowDroid specifically designed for confidential data leak detection in Android applications. It is lifecycle-aware, and implements context- and field-sensitive information flow analysis building on an IFDS framework [34].

Taint analysis detects confidential data leaks as propagation of "tainted" information through the system from so-called *sources* that introduce confidential data into the system to the *sinks* that expose the data to an unprotected environment. In the scope of our approach, developer is only required to indicate the sources of confidential data in the application through program annotations. Any method that reads and returns confidential data is a *source*.

In the *Google Authenticator* example, an engineer needs to protect the security key entry and manipulation, and indicates two sources – `intent.getStringExtra("SCAN_RESULT")` for obtaining QR-code scan result and `getEnteredKey()` for getting a manually entered key. These sources will be transformed into a TEE-controlled virtual keyboard service to manually enter the key, and a secure QR scanner service that uses a TEE-controlled camera, respectively.

A *sink* in our approach is defined for several situations where confidential data "extends" beyond the control of the application. In other words, a sink writes confidential data into a resource that can be accessed or controlled outside the application. In Android OS such resources include screen, network connection, storage, etc. In addition, specific to Android OS, the data extends beyond the control of the app when one component (such as *Activity*) of the app communicates with another component of the same app, which is not safe – any data passed to the rich OS can be leaked to unauthorized users.

For *Google Authenticator*, the taint analysis identifies that the security key flows through six application classes; from the sources to a local app database and then, depending on the scenario, through an OTP value, to the sinks – a text field to be displayed on a screen, or to the system clipboard.

We then use the analysis results in the application partitioning algorithm represented by phase ② in Figure 2. The algorithm implements our observation that in real-world applications a large portion of statements do not process data, but only transfer data along secure information flow paths. For instance, in *Google Authenticator*, a key read-store flow traverses seven methods in three classes, where only one method `getEnteredKey()` processes the data, while none of the other methods operates on the confidential data. Our insight is that such a "transfer" code does not require transformation, can remain in rich OS, and operate on surrogate data (unique opaque references), while supporting the correct application control flow and context sensitivity.

Another observation that supports our approach indicates that statements operating on confidential data can be grouped together when extracted into Trusted Applications. This is implemented in the phase ③ of the approach. For instance, the core functionality of *Google Authenticator* for OTP generation combines a set of transformation operations that hash the internal app state value, truncate the result and pad the output value. These consecutive statements can be grouped and extracted to TEE as a single code segment. Our approach leverages heuristics for grouping operations on confidential data.

Finally, our partitioning framework identifies corner cases where segments of code cannot be automatically extracted and reports them to the engineer. These include code segments that manipulate OS-dependent code, confidential operations with overlapping contexts that cannot be isolated properly, and code fragments control-dependent on confidential data.

By applying our approach to protect the security key manipulation in *Google Authenticator*, we extract eight code fragments (including fragments for key decoding, generation of message authentication code or MAC, key manipulation and display) with a total of 35 LOC, sufficient to protect the entire confidential data flow, instead of analyzing and transforming more than 1.6 KLOC related to the flow in the original application.

The final phase of the approach concerns TEE-specific engineering efforts for integrating Trusted Applications into Android OS and TEE (phase ④ in Figure 2). In this phase, our approach assists an engineer in transforming code fragments to native TEE Commands by automatically generating code with TEE API calls for establishing communication and parameter passing between normal and secure worlds.

# 4. PARTITIONING FRAMEWORK

Our approach starts with taint analysis enhanced with annotation of taint-propagating statements with contextual information (Section 4.1). We classify the annotated statements and capture a subset of the statements that will form a secure partition to be deployed on TEE (Section 4.2). In this process we identify groups of statements (Section 4.3)

and resolve corner cases (Section 4.4). To maintain the flow of data through "transfer" statements (statements that pass confidential data without modification), we substitute confidential data references with opaque references in the transformed application (Section 4.5).

## 4.1 Capturing path and context information

For our partitioning goal, we determine all the statements that transfer confidential data from one variable to another in a given context across reachable paths. To that end, we employ taint analysis and extend it to precisely capture contextual information by annotating statements that propagate taints from sources to sinks. Our analysis intercepts the results of flow function computation during taint analysis and registers contextual information including originating context and the corresponding source of the taint.

We encapsulate taint information for a taint-propagating statement in a tag $t_{\langle source,successor,type\rangle}$, where *source* is an incoming tainted variable (predecessor flow fact abstraction), *successor* is a variable propagating taint further, and *type* is an auxiliary entity describing the taint propagation condition for a given type of statement. We distinguish 27 types of taint-propagating statements (e.g., `ASGN_INSTANCE y=x.f`; `ASGN_ARRAY_REF y=x[i]`) used to classify transfer and privileged statements. *Source* and *successor* fields of a tag are used with alias analysis to ensure that we work only with explicit taint propagation.

It is not sufficient to store a single instance $t$ per statement, as a single statement may be traversed by the taint analysis with more than one source abstraction and in multiple method contexts. To that end, for each statement we record a set $T$ of all tags $t$ that corresponds to all the distinct traversals of the statement by the taint analysis.

We assure context-sensitivity of our analysis by collecting and matching the context of tagged statements with the contexts (incoming flow fact abstractions) of their enclosing methods. We define *method context* as a set of flow facts $D_{proc}$, such that for any tainted statement $n$ in a method *proc* there is a path edge[2] sourced at the exploded supergraph node corresponding to the entry point $s$ of *proc* and terminating at the given statement $n$: $\langle s_{procOf(n)}, d_1\rangle \rightarrow \langle n, d_2\rangle$, where flow fact $d_1 \in D_{proc}$, and $d_2$ is the new discovered flow fact abstraction at $n$. As $s_{procOf(n)}$ is uniquely identified by $n$, we only need $d_1$ to identify the path edge. In other words, different sets of flow facts $D_{proc}$ at the start of the method denote that the method is reached from different sources or different contexts.

We annotate statements that are both reachable in the taint analysis and have outgoing edges from the incoming flow fact abstraction to the other flow facts. To capture the method context information for each statement $n$, we associate each $d \in D_{proc}$ with a subset of tags $T_n \subseteq T$ that denote taint propagation in a given context.

## 4.2 Identifying candidate program segments

Our algorithm for selection of candidate program segments is shown in Algorithm 1. It takes as input an inter-procedural control-flow graph (ICFG) $G$, annotated representation of program statements (nodes) with flow fact information, lists of sources $S$ and sinks $K$. Starting from a method that contains an invocation of a source, it traces methods that propagate taint in both "forward" and "backward" directions, classifies and records candidate confidential statements.

**Algorithm 1** Analysis of candidate program segments

**Input:** $S$ – list of sources; $K$ – list of sinks; $G$ – interprocedural CFG; $M$ – worklist of methods;
**Output:** $OUT$ ▷ output is a map of candidate privileged stmts and associated input/output taint sets

1: $M \leftarrow \emptyset$; $M_{cache} \leftarrow \emptyset$
2: **for** $s$ in $S$ **do**
3: $M \leftarrow M \cup \{methodOf(s)\}$ ▷ Initialize worklist of methods
4: **while** $M \neq \emptyset$ **do**
5: $m \leftarrow pick(M)$
6: **if** $m \notin M_{cache}$ **then**
7: $D_m \leftarrow getMethodContext(m)$
8: **for** $stmt$ in $m$ **do**
9: $T_{stmt} \leftarrow getTags(stmt)$
10: **if** $isAnnotated(stmt) \wedge (\exists t \in T_{stmt} : D_m \Rightarrow t)$ **then**
▷ Process tagged statement with matching method context:
11: $OUT \leftarrow OUT \cup \{\text{processStatement}(stmt, m)\}$
12: $M_{cache} \leftarrow M_{cache} \cup m$
13: $M \leftarrow M \setminus m$
14: **procedure** processStatement($n, m$)
15: $P_n \leftarrow getInTaintSetOf(n, D_m)$
16: $R_n \leftarrow getOutTaintSetOf(n, D_m)$
STAGE 1: Extend the worklist
▷ Transfer call statement with a tainted parameter:
17: **if** $isCallStatement(n) \wedge (params(n) \cap P_n \neq \emptyset)$ **then**
18: $M \leftarrow M \cup \{getCallee(n, G)\}$ ▷ add callee to the worklist
19: **return** $\emptyset$
▷ Returning taint – add callers of m to the worklist:
20: **if** $isExitStatement(n)$ **then**
21: $M \leftarrow M \cup \{callersOf(m, G)\}$
22: **return** $\emptyset$
▷ Taint flows to a field variable – add callers of class methods to the worklist:
23: **if** $\exists r \in R_n \wedge isFieldVar(r)$ **then**
24: $c \leftarrow getDeclaringClass(r)$
25: **for** $m$ in $getMethodsOf(c)$ **do**
26: $M \leftarrow M \cup \{callersOf(m, G)\}$
▷ Source stmt taints parameters of enclosing method – add callers of m to the worklist:
27: **if** $(n \in S) \wedge (D_m \neq \emptyset)$ **then**
28: $M \leftarrow M \cup \{callersOf(m, G)\}$
STAGE 2: Record privileged statement
29: **if** $isPrivilegedStatement(n)$ **then**
30: **return** $(n, R_n, P_n)$
31: **else** ▷ transfer statements are not added
32: **return** $\emptyset$

The algorithm maintains a list of taint-propagating methods $M$. For each annotated statement in a method on a taint-propagating path, it first detects successive propagator methods to be added to the worklist (*Stage 1* in Algorithm 1). This is followed by *Stage 2*, where the annotated statement is classified and, potentially, recorded together with the sets of incoming and outgoing taints.

Taint can be propagated "forward" and "backwards", as exemplified in the implementation of taint analysis in FlowDroid. Forward analysis follows taint propagation along the call graph edges, while backward analysis traces taint propagation through statements at which tainted values are assigned to the heap, i.e., to fields or arrays, and searches backwards for aliases of these values. Our algorithm traces caller-callee relationships forward, and `return`, instance field, array, and container assignments backwards (*Stage 1*).

The algorithm descends into methods following taint from a method of a given source and checks generated taint abstractions. Upon reaching the next method, it resolves whether tainted abstractions produced from a given source are explicitly propagated to the statements in the body of this

[2]A *path edge* summarizes the effect of flow functions on a supergraph from the start node of a method $s_{procOf(n)}$ to a statement $n$ [34]. It forms the suffix of a realizable tainted path from the application entry point to the statement.

method. At each stage of the algorithm we use alias analysis and resolve variable access path to ensure that we consider only statments with explicit taint propagation. *Access path* models taint propagation to object fields [36] and is symbolically represented as $x.f.g$ or abbreviated as $x.f^k$ with $k = 2$ denoting a path of length two rooted at $x$. The notation allows us to split field accesses as $x.f^k = x.f^m.f^p$, where $k = m + p$.

We preserve deep field-sensitivity in the analysis ensuring that only statements that directly (explicitly) use the tainted access path $x.f^m.f^p$ (an access path with $m + p$ fields) or a longer access path accessible through $x.f^m.f^p$ are processed as candidate statements. Statements that only refer to $x.f^k$ where $k < (m + p)$ correspond to implicit taint propagation that does not reveal confidential data references, and need not be considered as far as moving to secure world is concerned.

The following listing highlights the basic cases of explicit and implicit taint propagation:

```
// Example 1: Explicit taint propagation:
y = x.f // where x or x.f is tainted
// Example 2: Implicit taint propagation:
y = x   // where x.f is tainted
// Example 3: Implicit taint propagation:
y.f = x // where x.f is tainted
```

In *Stage 2* of the algorithm, we classify taint-propagation statements. We distinguish *privileged* and *transfer* taint-propagation statements. Statements that operate on, modify and multiply instances of confidential data are *privileged*. Conversely, *transfer* statements access confidential data and propagate it without modifying it. Examples of transfer statements include: method invocations that pass tainted data into a body of a method through a function parameter: `update(secret)`; assignment statements of a form `x = secret`, where tainted variable `secret` is not modified; return statements in the form `return secret`.

*Privileged* statements modify the value of a passed tainted data and/or derive new instances of tainted data. In Algorithm 1, the classification of taint-propagation statements is incorporated in a condition $isPrivilegedStatement(n)$ that expands into the following set of conditions:

$$(isAssign(n) \wedge hasOp(getRHS(n))) \vee isArrayAssign(n)$$
$$\vee isConditional(n) \vee (n \in S) \vee (n \in K) \vee isWrapperCallStatement(n)$$

Each of the clauses helps to identify different kinds of privileged statements. These include arithmetic, bitwise and bit shift operations on primitive data types to derive new tainted values (`x = taint + 42`) and array manipulation operations (`a[i] = taint`). Our algorithm checks these cases with the conditions $isAssign(n) \wedge hasOp(getRHS(n))$ and $isArrayAssign(n)$ respectively. To ensure secure information flow, we classify statements that assign elements of tainted arrays as privileged (`taint[i] = y`). We also classify source and sink invocations as privileged (clauses $(n \in S)$ and $(n \in K)$). Condition $isConditional(n)$ checks for a special case of control-dependent taint that we discuss in Section 4.4.

Not all operations in object-oriented software can be (or shall be) traced to the lowest level of manipulation of primitive data types. Such operations include external and library functions whose code may not be available, standard Java and Android functionality (for instance, container classes are commonly used and difficult to analyze), GUI functions, etc. FlowDroid abstracts taint propagation for these operations using so-called *taint wrapping* or explicit taint propagation rules. We classify statements involving these operations as privileged ($isWrapperCallStatement(n)$ condition).

```
1  public void process(View v) {
2   String secondPin = pincode.getText().toString();
3   if (!firstPin.equals(secondPin)) { // TEE_cmd()
4       /* ... */
5       _showErrorView(details); // implicit flow
6       return;
7   }
8   _hideSoftKeyboard(pincode);
9   /* ... */
```

**Figure 3: Implicit flow in *tiqr* pin verification**

Note that the same set of statements operates on both benign and confidential data in different contexts or along different execution paths. Our technique provides a feedback to an engineer that can either indicate that a potential source of sensitive data is not included in the analysis, or suggest that overlapping application paths have to be decoupled.

## 4.3 TEE Command creation by grouping

The two competing requirements imposed by TEE include minimization of TCB and minimization of the communication overhead between app in normal world and a trusted application. Extraction of minimal code fragments ensures small TCB, while extraction of multiple fragments contributes to increasing the communication overhead. We develop a heuristic to ensure appropriate granularity for each TEE Command through consolidation of temporally close secure partitions.

We consider each privileged statement as a candidate for being extracted as a distinct TEE Command. Our heuristic applies intra-procedurally to aggregate privileged statements within the scope of enclosing methods into groups of statements forming distinct TEE Commands.

We aggregate both consecutive privileged statements and non-adjacent groups of privileged statements (separated by other constructs). To that end, we consider sets of variables tainted within a group and types of statements that separate the privileged statements such as *assignment* or *conditional*. We aggregate normal non-privileged statements in the groups, where the type of separating statements determines whether we can aggregate the adjacent groups. To preserve the application semantics we do not aggregate statements separated by a conditional statement, nor by Android OS methods such as `startActivity(intent)` that can affect the control flow of an app. The group size is thus bounded either by the enclosing basic block (for groups separated by conditional statements) or by an enclosing method.

## 4.4 Control-dependent taint and implicit flows

Conditional expressions controlled by confidential data may lead to *implicit flows* – conditional executions implicitly leaking confidential data by updating non-privileged data in a conditional structure. Consider an example of an implicit flow in the code of *tiqr*, an open source solution for user authentication on web applications using smart phones [38]. Figure 3 shows an abridged code for an implicit flow in *tiqr* in an enrollment pin verification activity. Checking the correctness of a pin entry (confidential data) affects the control flow of the enclosing method `process()`—an error dialog is shown to a user if two pin codes do not match—thus leaking information about privileged data.

Implicit flows directly affect the granularity of application partitioning in our approach, and, in some cases, represent the limits of our automated solution w.r.t. complete pro-

tection of confidential data. Complete protection requires deployment of all implicitly tainted data flows as secure TEE Commands. This includes protection of a complete control structure or a complete method, if the control structure contains `return` statements. In the example in Figure 3, a complete protection of the implicitly tainted data would require method refactoring to obtain a single method exit point, and deployment of a code for visualization and handling of an error dialog as a TEE Command.

In cases when complete protection is possible, we classify all statements in a conditional structure as privileged, and extract this group of statements into a single TEE Command. In many cases complete protection is infeasible or detrimental due to high contribution of an implicitly tainted code to the TCB size, or due to the presence of a rich OS-dependent code (e.g., GUI, database access) that cannot be moved to TEE without breaking application semantics. In these cases, our technique provides engineer with a feedback highlighting an implicit flow that has to be protected manually.

It is possible to release an implicitly tainted information and support our automated solution by declassifying a boolean result of a tainted conditional expression and extracting the condition as a TEE Command invocation. In the example in Figure 3, expression `firstPin.equals(secondPin)` can be substituted for an invocation of a TEE Command without revealing the confidential data. Yet, out of the three basic types of control structures such as *if-then-else/loop/switch*, only *if-then-else* is practically amenable to this solution.

## 4.5 Unique opaque references

To support the preservation of transfer statements in normal world we provide an *opaque reference* mechanism to securely transfer references to confidential data within normal world and from normal to secure world. Opaque references enable context-sensitive addressing of confidential data from normal world in cases when privileged statements can be reached from different contexts or with data propagated from different sources.

In our approach an *opaque reference* is an object reference that points to a unique Java object of a required type, whereas object's unique hash code serves as a key to a hashtable of actual confidential data references stored in TEE. We create a reference by allocating a new unique Java object of a required type. Each opaque reference produced in a specific context uniquely identifies this context to TEE.

We generate opaque references of types expected by original implementation thus avoiding compile and runtime errors. To uniquely identify primitive types, we apply minor code refactoring on the original application and substitute tainted primitive variables with objects of primitive wrapper classes.

Opaque references do not conflict with polymorphic method invocations. This is because polymorphic method invocations with tainted base objects are marked as privileged by our approach and deployed in TEE Commands. Thus, the runtime type of a base object (its opaque reference) does not affect the control flow of the application.

# 5. IMPLEMENTATION

The main components of our system are built as extensions to the FlowDroid analysis on top of the program analysis framework Soot [27]. Figure 4 shows a general view of the components of our system. In addition to the extensions, our framework provides a code generation facility and Wrapper to

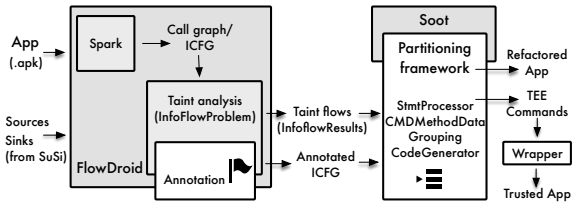


**Figure 4: System implementation**

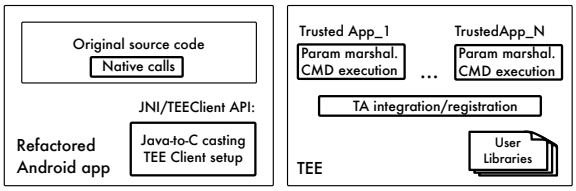


**Figure 5: Generated and transformed source code**

generate native TEE Client and Internal API calls used for registration of trusted application and establishing communication between a refactored app and trusted applications. A complete template of generated and modified source code of a transformed application is schematically shown in Figure 5.

Our framework works with a developer-generated list of confidential data sources and a list of sinks. To generate a list of sinks, we use a machine-learning approach *SuSi* that classifies Android sources and sinks extracted from source code using the semantic and syntactic features [33]. We use *SuSi* to generate a list of sinks for specifying standard shared resources in Android (files, network connections, etc.) and the sinks related to app lifecycle that persist and pass information within an app (between app components) and between apps. In addition, we treat declassifier methods (e.g., methods that return encrypted data) as sinks.

## 5.1 Tagging statements across reachable paths

Our implementation annotates program statements during taint analysis using tagging functionality of the Soot framework. The granularity of our analysis is at the level of program statements in Jimple intermediate representation used by Soot [37]. We integrate the annotation in the `InfoflowProblem` implementation of FlowDroid and annotate statements during flow function computation with free-form labels that implement the `Tag` interface. Using these tags we capture path and context information as described in Section 4.1.

## 5.2 Partitioning framework

We retrieve the taint analysis results by implementing the `ResultsAvailableHandler` interface of FlowDroid and post-process the statements that we have tagged. The post-processing implements Algorithm 1 where tagged statements are classified, grouped, and transformed. We abstract all confidential variables and operations on confidential variables as static methods in a separate program partition shown as TEE Commands in Figure 4. These static methods are later transformed to native Trusted Applications with the help of the Wrapper. The Wrapper also generates the code for the allocation and association of opaque references to each of the confidential variable.

For groups of statements, we use their input/output variable sets and generate single TEE Command invocation calls

for each group. We then automatically substitute the groups for invocations of TEE Command methods. As a result, we produce a modified version of the original application shown as Refactored App in Figure 4.

## 5.3 Generation of TEE-specific code

The transition from extracting secure partition to its deployment as TEE Commands is not trivial and our framework assists a developer in this process. TEE Commands are fragments of C code that use standard C library available on TEE, TEE Internal API (including Trusted Storage, Cryptographic operations, Timer API, etc.), and user libraries that developer can integrate in the TEE code base.

Our approach does not automatically translate the privileged instructions from Java to native TAs. Once the engineer transforms the Java priviledged instructions to C code, our framework automatically generates TEE-specific code with Wrapper. In particular, the Wrapper generates TEE Command signatures, TEE Internal API invocations for command identification, parameter marshalling, and memory allocation and access (Param. marshal. and CMD execution in Figure 5).

On the normal world side of the app, the Wrapper generates the TEE Client API code for invoking commands from normal world. It generates code for instantiating TEE Context — an abstraction of the logical connection that exists between the client Android application and TEE. Inside the TEE Context that the app has created, it initializes TEE Session — an abstraction of the logical connection between the app and a specific Trusted Application identified by a command ID. This code is schematically shown in Figure 5 as TEE Client setup.

Operations in TEE Client setup open a TEE Session or invoke a command of a Trusted Application. These operations may carry additional payload, stored inside a TEE Operation data structure that permits a maximum of four parameters. To overcome the restriction, we serialize the parameters inside the input memory reference and deserialize them in the Trusted Application command processing function. To support opaque references, we generate a Hashmap structure on the trusted partition in TEE that keeps a mapping between opaque references and actual values in its secure memory.

The deployment of TEE Client API chain of calls inside an Android app necessitates interfacing native C Trusted Application invocation with Java Native Interface (JNI) to be called from the Java code of the app. Thus, we wrap the command invocation with a JNI C method (shown as Native calls and Java-to-C casting in Figure 5). We automatically generate respective headers and integrate the generated JNI C methods as a native library that we generate using Android Native Development Kit (NDK) [1].

# 6. EXPERIMENTAL EVALUATION

We evaluate our approach with six real-world applications and a set of micro-benchmarks. We demonstrate that our approach successfully partitions applications with simple and complex flows of confidential data, and generates TAs that exhibit low TCB and realistic execution time compared to the original applications.

## 6.1 Experimental setup

We explore the capabilities of a commercial Trusted Execution Environment, SierraTEE [9], on Microvision MV4412 board [6] with ARM Trustzone. The MV4412 board features Samsung Exynos4412 system-on-chip with ARM Cortex-A9 quad core running at maximum frequency of 1.4GHz, 4 GB eMMC, and 1 GB LPDDR2 memory. We installed POSIX-compliant SierraTEE as secure kernel and Android Version 4.0.3, Kernel 3.0.15 as the rich OS.

**Table 1: Micro-benchmarks – results**

| **SecuriBench** | Correct/Total | **DroidBench** | Correct/Total |
|---|---|---|---|
| Aliasing | 5/5 | Aliasing | 1/1 |
| Arrays | 6/6 | ArraysAndLists | 2/3 |
| Basic | 30/40 | FieldAndObjectSens | 7/7 |
| Collections | 11/11 | GeneralJava | 23/23 |
| DataStructures | 5/5 | ImplicitFlows | 1/2 |
| Factories | 3/3 | **Control-dependent** | Correct/Total |
| Inter | 11/12 | DecisionProtecSimple | 9/12 |
| Pred | 6/8 | DecisionProtec | 6/8 |
| StrongUpdates | 4/4 | | |

## 6.2 Micro-benchmarks

We highlight the applicability and the correctness of the approach in a variety of situations by applying it to a set of standard Android benchmarks *DroidBench* [24] and an adapted set *SecuriBench* [30]. These benchmarks are designed to check taint analysis for different cases of data flow arising in secure context. We have adapted *SecuriBench* to Android by integrating original Java test cases in Android applications. We have also extended the test set with two of our own benchmarks to evaluate the partitioning of applications with control-dependent flows, where we extract the decision part of the control structure as a TEE command. We use FlowDroid taint analysis from April 1, 2015.

Table 1 summarizes the results of automated partitioning applied to the micro-benchmarks. Partitioned apps have not been deployed to TEE. The total number of cases of confidential data flow from sources to sinks for each benchmark obtained through taint analysis is reported (*Total*). We then apply our prototype framework to all these cases, manually check the partitioning, and report the number of cases where the resulting transformation is successful (*Correct*).

Our approach can successfully partition and transform 86% of the cases across all the benchmarks. The analysis and transformation time for all the benchmarks is only a few seconds. We observe that the invalid transformations are due to the limitations of the current prototype. In particular, the prototype is not fully supporting transformation of tainted conditional structures. The largest share of inapplicable cases (ten cases in Basic of *SecuriBench*) map directly to the extraction of control-dependent taint. Our approach can extract only some cases of control-dependent taint shown in Table 1. Other cases of taint propagation are covered exhaustively by our approach. We are implementing minor modification to the prototype to avoid these issues.

## 6.3 Case studies

We select six widely-used open-source applications as case studies. *Google Authenticator*, *tiqr*, *OpenKeychain*, *card.io*, *Hash it!*, and *Pixelknot* provide enhanced security of user data, communication and/or authentication processes. We apply our approach to each case study to generate candidate program partitions. We then transform and deploy the partitioned applications to Android OS and SierraTEE. Trusted Applications are loaded to SierraTEE at compile time.

**Google Authenticator.** Introduced in Section 3, the *Google Authenticator* (GA) generates a six- to eight-digit One-Time

Passwords (OTPs) that users supply in addition to their user name and password to log into Google services or other sites [4]. GA uses a security key provided by Google (scanned or manually entered) to generate and cryptographically sign the generated OTPs. The secure partitions extracted by our approach correspond to reading and signing the secure key for protecting generation of counter- and timer-based OTPs. In addition to protecting OTP generation, we have integrated in secure partition a standard Virtual Keyboard service of SierraTEE for secure manual key entry, and a QR code parsing library *Quirc C* [18] for secure key entry by means of QR code scanning. To support cryptographic operations in TA, we replace the `javax.crypto.Mac` library used by original app with an OpenSSL counterpart `HMAC_*`, and configure SierraTEE to provide OpenSSL library support.

**tiqr.** *tiqr* is an open source authentication solution for smart phones and web applications [38]. It is based on Open Standards from the Open Authentication Initiative (OATH) [14]. It performs challenge/response authentication using QR codes. Similar to the GA app, we have integrated in a secure partition the Virtual Keyboard service of SierraTEE for secure PIN entry, and a QR code parsing library *Quirc C*.

Our approach extracts four TEE commands for *tiqr* that together constitute an authentication service for logging into a web-site. `CMD1` scans a QR code with an encoded authentication challenge projected by a web-site and returns an opaque reference to the scanned result. `CMD2` parses the result and based on the challenge from the decoded QR code, chooses user enrollment or authentication thus affecting the control flow of the app. This is followed by user authentication on the phone by PIN entry where `CMD3` is called to enter and validate the PIN entry using virtual keyboard. Depending on the control decision in `CMD2`, `CMD4` extracts enrollment or authentication URL from the decoded QR code and sends it to the server for validation and login authorization.

**OpenKeychain.** *OpenKeychain* (*OK*) [7] is an open-source digital key management app based on OpenPGP standard [20]. It supports encryption, decryption, signature generation, signature verification of files/text, and provides key exchange via QR or NFC with secret keys stored on Yubikey devices.

In this case study we protect the generation of 2048-bit RSA key that we deploy as a TEE Command `genRSA`, and encryption/signing of the user-provided text with a securely stored private key `signRSA`. The app first calls `genRSA` to create the confidential data – the key. The key is then used in sign operation (a sink) for text encryption. The two generated TEE Commands pass an opaque reference to the private key stored in TEE. The `signRSA` command takes a byte array input and the opaque reference, encrypts the text in TEE, and returns the encrypted text as a byte array. As in the case of GA, we have configured SierraTEE to use OpenSSL in place of *Spongy Castle* Android cryptography APIs.

**card.io.** *card.io* (*CI*) is a simple credit card scanning interface for mobile apps [2]. The two generated TEE commands capture the user input of a card or cvv number on the TEE side and return it to the normal world.

**Pixeknot.** *Pixeknot* (*PK*) implements steganographic matrix encoding algorithm to help users to hide confidential text-based messages in photographs and share them across trusted channels [8]. The four generated commands include: `CMD1`: captures the secret message to be hidden in an image and stored on TEE. `CMD2`: captures the password/salt used for encryption and stored on TEE. `CMD3`: extracts the secret message on the TEE side, encrypts it using AES-GCM-256, and returns the encrypted text to the normal world. `CMD4`: takes the extracted text as input, decrypts the message on the TEE side and returns plain text to the normal world.

**Table 2: Client code and Trusted Computing Base. CCF = Confidential code fragment; JNIC = JNI + Java-to-C code; TCAC = TEE Client API code; TCC = TEE Command code; PM&TIAC = Param. marshal.+ TEE Internal API code; LIB = User or external library.**

| Trusted App Command | Original app | | Normal World | | Secure World | | |
|---|---|---|---|---|---|---|---|
| | Size (KLOC) | CCF (LOC) | JNIC (LOC) | TCAC (LOC) | TCC (LOC) | PM&TIAC (LOC) | LIB (KLOC) |
| GA TOTP | 3.7 | 3 | 49 | 113 | 6 | 218 | 134.9 |
| GA HOTP | - | 6 | 9 | 95 | 8 | 143 | 134.9 |
| tiqr CMD1 | 6.1 | 8 | 15 | 121 | 11 | 250 | 1.9 |
| tiqr CMD2 | - | 1 | 20 | 116 | 6 | 260 | n/a |
| tiqr CMD3 | - | 115 | 5 | 116 | 1 | 40 | 1.37 |
| tiqr CMD4 | - | 1 | 20 | 116 | 6 | 260 | n/a |
| OK genRSA | 57 | 1 | 31 | 125 | 24 | 210 | 131.7 |
| OK encRSA | - | 1 | 48 | 125 | 24 | 232 | 131.7 |
| CI CMD1 | 15 | 30 | 5 | 90 | 5 | 210 | 1.37 |
| CI CMD2 | - | 33 | 5 | 90 | 5 | 210 | 1.37 |
| PK CMD1 | 5 | 1 | 5 | 90 | 5 | 210 | 1.37 |
| PK CMD2 | - | 1 | 5 | 90 | 5 | 210 | 1.37 |
| PK CMD3 | - | 1 | 42 | 120 | 76 | 290 | 131.7 |
| PK CMD4 | - | 1 | 52 | 130 | 120 | 260 | 131.7 |
| Hash it! | 6 | 4 | 49 | 114 | 6 | 218 | 131.7 |

**Hash It!** *Hash It!* generates site-specific passwords derived from a secret master key [5]. The extracted TEE command deploys to TEE a cryptographic message authentication code (MAC) function to derive site-specific passwords.

### 6.3.1 Developer Effort and TCB size

Table 2 summarizes the contribution of the commands to the Trusted Computing Base (TCB) size in SierraTEE and the changes to the client code.

As we can see from Table 2, the original code size for our case-study applications is quite large ranging from 3.7 KLOC to 57 KLOC. However, the confidential code fragments identified by our analysis engine are pretty small ranging from 1 LOC to at most 115 LOC for a command. It is clear that the developer would need to spend significant effort to manually identify these code fragments — a burden that is completely relieved through automation.

The columns under *Normal World* correspond to the components required to integrate TEE commands with an Android app. We have highlighted these components schematically in Figure 5 using thick borderlines. For each TEE command, an app needs JNI code to pack and unpack parameters from Java to C and back (JNI Java-to-C). The normal world also includes TEE command setup using TEE Client API and TEE client headers that hold structure definitions for command invocation parameters. TEE Client API code is automatically generated by our framework.

The components under *Secure World* contribute to TCB size. These include parameter marshalling (Param. marshal.) and command setup using TEE Internal API, command registration code (a constant of 100 LOC for each command, not shown in Table 2), the core code that performs the function corresponding to the command (TEE Command code), and user/external libraries registered to TEE. These contribute to at most 380 LOC to TCB for an application, which is really small compared to the 41.6KLOC TCB of Sierra OS.

**Table 3: TEE Command execution time. Mean values with standard deviations in parentheses.**

| Trusted App Command | Orig. app exec. | JNI copy exec. | TEE Command exec. |
|---|---|---|---|
| Concat | 13 $\mu$s (0.9) | 9 $\mu$s (15) | 9 $\mu$s (10) |
| Multiply | 140 $\mu$s (10) | 30 $\mu$s (11) | 30 $\mu$s (10) |
| GA TOTP | 640 $\mu$s (107) | 40 $\mu$s (4) | 85 $\mu$s (18) |
| GA HOTP | 600 $\mu$s (28) | 40 $\mu$s (3) | 70 $\mu$s (20) |
| tiqr CMD1 | 14 $\mu$s (3) | 13 $\mu$s (1) | 250 $\mu$s (35) |
| tiqr CMD2 | 21 $\mu$s (6) | 13 $\mu$s (1) | 220 $\mu$s (10) |
| tiqr CMD3 | 2.5 $\mu$s (0.4) | 0.8 $\mu$s (0.04) | 78 $\mu$s (5) |
| tiqr CMD4 | 19 $\mu$s (4) | 10 $\mu$s (0.5) | 220 $\mu$s (14) |
| OK genRSA | 2.8 s (1.8) | 0.6 s (0.3) | 0.5 s (0.3) |
| OK encRSA | 0.8 s (0.04) | 0.034 s (0.0009) | 0.1 s (0.001) |
| CI CMD1 | 3.8 $\mu$s (0.8) | 0.7 $\mu$s (0.03) | 78 $\mu$s (5) |
| CI CMD2 | 3.2 $\mu$s (0.7) | 0.6 $\mu$s (0.06) | 79 $\mu$s (5) |
| PK CMD1 | 3.2 $\mu$s (0.5) | 0.9 $\mu$s (0.06) | 86 $\mu$s (6) |
| PK CMD2 | 4.6 $\mu$s (0.4) | 0.7 $\mu$s (0.03) | 80 $\mu$s (5) |
| PK CMD3 | 1.99 s (0.0001) | 26 $\mu$s (3) | 280 $\mu$s (34) |
| PK CMD4 | 2.11 s (0.0002) | 27 $\mu$s (5) | 267 $\mu$s (32) |
| Hash it! | 557 $\mu$s (61) | 27 $\mu$s (5) | 71 $\mu$s (10) |

Consider, for instance, tiqr `CMD3` in Table 2. The original confidential code fragment with 115 LOC involves two activity classes for reading and verifying user pin entry. The corresponding TEE version reads the pins using virtual keyboard and compares them using 1 LOC. The contribution to TCB mostly comes from parameter marshalling (40 LOC) and external library usage (1.37 KLOC Virtual Keyboard service) reaching 1.41 KLOC. Nevertheless, most of generated TEE Commands contribute more lines of code in C than their original confidential code fragments in Java. For instance, `CMD4` in *Pixelknot* substitutes a single message decryption statement in Java with 120 LOC on TEE.

Consideration of transfer statements gives us up to two orders of magnitude reduction in LOC in TEE Command Code compared to the pure taint analysis that would extract all the tainted statements potentially breaking app semantics, and contributing to high TCB.

Note that the only code that has to be manually written by the developer is JNI Java-to-C in normal world and TEE Command Code in secure world. This is because we do not handle automated translation from Java to C for TEE Commands. Fortunately, both these components require minimal effort as can be seen from the table.

### 6.3.2 Runtime Overhead

In addition to the real-world case studies, we generate two simple applications to evaluate the runtime overhead of our approach. The first application *Concat* is a trivial TA that takes two characters as input and returns the concatenation of the two to accentuate the communication overhead between two worlds. The second application *Multiply* multiplies two $10 \times 10$ integer matrices and thus can differentiate computation time in TEE from communication overhead.

We compare the execution time of TEE command with the execution time of the original Java code and with a JNI C alternative. The JNI C code is executed in Android OS; but not deployed to TEE. We perform ten measurements for each case and present in Table 3 mean end-to-end execution time of single iterations of user scenarios followed by standard deviations in parentheses.

For individual command invocations, computation in TEE is up to an order of magnitude faster than the original application for pure computation (*Concat, Multiply, Google Authenticator*). This is not surprising, because execution of C code is usually faster than execution of Java code. For *OpenKeychain, Hash it!*, and *Pixelknot* `CMD3` and `CMD4`, the TEE version takes less time than original app because cryptographic operations are much faster in SierraTEE (C code) compared to Android. We have confirmed this by running the JNI C copy of the original code that shows comparable execution time to TEE. Services using virtual keyboard or reading/writing persistent objects (*tiqr, card.io, Pixelknot* `CMD1` and `CMD2`) take up to an order of magnitude more time in TEE than in the original application, because they require extra setup and checks on TEE.

Most of the overhead from the complete TEE solution is the penalty for setting up TEE context, establishing TEE session, and switching between normal and secure world. A context/session represents a logical connection between a client application and a specific TA. Multiple TEE commands can be issued inside an established session for the client application to access services provided by the TA. A command invocation in a session adds only $100\mu$s overhead on an average for switching between two worlds, while each new TEE context/session adds an average of 6.5ms overhead. These overheads get added over the computation time shown in Table 3. Note that since our example applications are reactive software and hence non-terminating, they do not have any concept of end-to-end delay.

## 7. THREATS TO VALIDITY

We highlight the limitations of the presented approach and threats to validity of our results.

As usual, with a different set of subjects the size of the TCB could be different. The underlying taint analysis in our approach, has some limitations in determining flow through recursive data structures as well as limited precision in distinguishing elements of arrays. In reporting experimental results in Table 1, the *Total* score differs from the results reported in the FlowDroid work [25]. The difference is due to our framework using a more recent version of FlowDroid and due to our adaptation of the benchmarks that affected the precision of FlowDroid analysis when applied to bytecode instead of source code.

There are also some limitations in transformation and partitioning of our approach. Despite contextual sensitivity, the approach does not handle situations when the same method is tainted from different contexts with different combinations of tainted method parameters. Currently, whenever we replace privileged statements with TEE Command invocations, we check that the tainted method-parameter-context combination is unique.

Finally, in generation of opaque references, not all objects of a required type can be generated. The approach instantiates classes with public constructors, whereas singleton or Android system classes, for instance, limit class instantiation.

## 8. RELATED WORK

**TrustZone and TEE infrastructure.** TrustZone security extensions isolate security-sensitive application logic from rich OS and other applications, and thus protect an application by deploying confidential partition to TEE. Main challenges in adopting TrustZone-based solutions include realizing approaches that maintain a small TCB size and enable seamless application development [23, 39]. Recent

effort to address these challenges includes general solutions in standardization of TEE interfaces and protocols [31,32,40], and custom solutions for TrustZone infrastructure [17, 35].

To facilitate seamless application development Marforio et al. implement secure enrollment protocols for TrustZone enabling service providers to associate the identity of a user to her device [31, 32]. Approach by Winter et al. develops a platform emulation tool based on QEMU platform emulator to help developers to design for TrustZone emulating ARM TrustZone platforms [40].

*TZ-RKP* approach implements a safe security monitor leveraging TrustZone architecture [17]. The monitor provides a real-time OS kernel protection by routing privileged system functions through secure world for examination. In contrast to our analysis-based approach that focusses on protection of individual applications and data, the *TZ-RKP* approach can be seen as a low-TCB system level solution.

A notable alternative to TEE is a custom Trusted Language Runtime (TLR) [35]. TLR is an infrastructure to run `.Net` applications using TrustZone extensions to isolate security-sensitive application logic. Provided with a partitioned application, TLR is able to interpret intermediate `.Net` code in its runtime engine. High-level language support on secure world ensures ease of development and verification on TLR. However, considering the restrictions of TLR that cannot access peripherals, the usability gain is uncertain since only small code fragments shall be deployed to secure world to preserve small TCB.

**Program partitioning.** To the best of our knowledge, there are no solutions available to secure Android applications for integration with TEE using automatic partitioning. However, there is a body of related work tackling program partitioning of other types of applications either for security policy enforcement in distributed computing or for privilege separation [15, 19, 21, 22, 26, 29, 41, 42]. Existing solutions attempt to compartmentalize application into partitions with different privileges, reduce the attack surface, minimize TCB and communication between parts of the applications with different privileges, and maintain secure information flow.

**Partitioning for privilege separation.** Partitioning for privilege separation prevents malicious exploitation of applications that typically run with maximum privilege by segmenting them into smaller components that can be granted minimal necessary privileges [15, 19, 26, 41]. A *ProgramCutter* collects information flow and privilege levels of program functions from dynamic traces keeping track of the size of program fragments [41]. It then applies *mincut* algorithm to partition program into function-level components with different privileges to minimize their size and communication.

A related approach *Privtrans* requires expert knowledge to specify privileged functions and variables [19]. It annotates the source code and partitions source program into only two components: a privileged program (the monitor) and an unprivileged program (the slave). *Privtrans* does not deal with complex features of the C language such as pointer arithmetic, function pointers and unrestricted type casting. The approach aims at minimizing the privileged code size, but not the slave-monitor communication overhead.

*Virtual Application Partitioning* enables developers to manage application attack surface by protecting the most exposed partition [26]. The approach automatically partitions software based on the intrinsic property of user authentication. Analysis of the application binary enables ranking of potential risks at specific points of application and selection of corresponding protection measures. The approach helps to reduce costs by enabling protection on demand.

*SeCage* approach by Liu et al. combines static and dynamic analyses to partition C applications w.r.t. sensitive data, and provides a supporting memory protection mechanism enabling a vitalization-based application protection [29]. The extracted partition is possibly incomplete aiming at reduction of TCB by protecting only the most commonly used sensitive functions discovered with dynamic analysis.

**Partitioning for secure distributed computation.** Another set of language-based techniques protects confidential data during computation in distributed systems containing mutually untrusted hosts [42]. Using annotations, information flow is constrained between trusted/non-trusted hosts through appropriate automatic partitioning. The approach was extended to tackle the challenge for secure web and database applications [21, 22]. These systems synthesize a good host assignment for each field or statement, and rely on type checking that lacks the context- and flow-sensitivity of data flow analysis. To alleviate the issue, systems are enriched with a client and a server runtime system to manage synchronization between the two programs, and correct for the assumptions of the static analysis. This increases the amount of code deployed to the secure server side. Moreover, these approaches allow for control of the program execution to be passed in both directions, whereas in a typical TEE setting, most of the program is executed in rich OS, and only small computations are invoked as commands in TEE.

# 9. CONCLUSIONS

In recent years we have witnessed the growth of technologies to support hardware enforced application protection. In this paper we propose to harness this technology without incurring high development costs by automatically adapting existing applications to the new technology. The proposed approach requires only a binary package of the original application, leverages state-of-the-art taint analysis, and automatically identifies and extracts code fragments relevant to manipulation of confidential data. These fragments, grouped and transformed, are deployed to Trusted Execution Environment with the help of our approach. The only non-automated step of the technique is the translation of the code fragments from Java to C. However, because the fragments are small, this manual effort is not overwhelming. The proposed approach provides fine-grained protection of real world Android applications. It produces a small Trusted Computing Base (TCB), incurs an adequate overhead, and enables application developers to adapt existing Android apps to take advantage of existing secure hardware / virtualization technologies.

# Acknowledgments

We thank Mingyuan Gao, Nithilan Meenakshi Karunakaran, and Manh Dung Nguyen for help with experimental setup. This research was partially supported by – (i) Singapore MoE AcRF grant T1 251RES1314, (ii) a grant from DSO National Laboratories, and (iii) the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

## 10. REFERENCES


[1] Android ndk toolset. https://developer.android.com/ndk/index.html.

[2] card.io. https://www.card.io.

[3] Global platform device specifications. http://www.globalplatform.org/specificationsdevice.asp.

[4] Google authenticator. https://github.com/google/google-authenticator-android.

[5] Hash it! http://android.ginkel.com/.

[6] Microvison co., ltd. microvision mv4412 board. http://www.boardset.com/products/products_v4412.php.

[7] Openkeychain. http://www.openkeychain.org/.

[8] Pixelknot. https://guardianproject.info/apps/pixelknot/.

[9] Sierraware: Sierratee trusted execution environment. http://sierraware.com/open-source-ARM-TrustZone.html.

[10] The MITRE Corporation: List of common vulnerabilities and exposures for all versions of Google Android. http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224/.

[11] GlobalPlatform Device Technology TEE Client API Specification Version 1.0 GPD SPE 007. Technical report, July 2010.

[12] GlobalPlatform Device Technology TEE Internal API Specification Version 1.0 GPD SPE 010. Technical report, December 2011.

[13] GlobalPlatform Device Technology TEE System Architecture Version 1.0 GPD SPE 009. Technical report, December 2011.

[14] Initiative for open authentication. http://openauthentication.org/specification, 2015.

[15] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 429–444, Bellevue, WA, 2012. USENIX.

[16] ARM. Arm security technology – building a secure system using trustzone technology. arm technical white paper. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[17] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 90–102, New York, NY, USA, 2014. ACM.

[18] D. Beer. Quirc. https://github.com/dlbeer/quirc/.

[19] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 57–72, 2004.

[20] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880: OpenPGP Message Format. Rfc 4880, RFC Editor, November 2007.

[21] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

[22] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM.

[23] J.-E. Ekberg, K. Kostiainen, and N. Asokan. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1497–1498, New York, NY, USA, 2013. ACM.

[24] C. Fritz, S. Arzt, and S. Rasthofer. Droidbench test suite. http://sseblog.ec-spride.de/tools/droidbench/.

[25] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2014.

[26] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis. Adaptive defenses for commodity software through virtual application partitioning. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 133–144, 2012.

[27] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[28] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, Nov. 1992.

[29] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *22th ACM Conference on Computer and Communications Security*, Denver, Colorado, US, October 2015.

[30] B. Livshits. Securibench micro test suite. http://suif.stanford.edu/~livshits/work/securibench-micro/.

[31] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. Secure enrollment and practical migration for mobile trusted execution environments. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, pages 93–98, New York, NY, USA, 2013. ACM.

[32] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. Smartphones as practical and secure location verification tokens for payments. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'14, 2014.

[33] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. *Network and Distributed System Security Symposium (NDSS)*, 2014.

[34] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[35] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 67–80, 2014.

[36] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 210–225, Berlin, Heidelberg, 2013. Springer-Verlag.

[37] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[38] R. M. van Rijswijk and J. van Dijk. tiqr : a novel take on two-factor authentication. In *Proceedings of LISA '11: 25th Large Installation System Administration Conference*, pages 81–97, Boston, MA, 2011. USENIX Association.

[39] A. Vasudevan, J. M. McCune, and J. Newsome. *Trustworthy Execution on Mobile Devices*, volume 8 of *SpringerBriefs in Computer Science*. Springer, 2014.

[40] J. Winter, P. Wiegele, M. Pirker, and R. Tögl. A flexible software development and emulation framework for arm trustzone. In *Proceedings of the Third International Conference on Trusted Systems*, INTRUST'11, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[41] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 323–333, Nov 2013.

[42] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, Aug. 2002.