Automated Partitioning of Android Applications for Trusted Execution Environments

Demil Omerovic

[Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, Abhik Roychoudhury]
• Increase for services like
  • online banking, premium content access, enterprise network connection,…

• Adapting open software platforms, installing 3rd party applications
  • Potential entry point for attackers

• Countermeasure -> security through HW protection

• ARM TrustZone
  • TEE
  • TrustZone technology
  • HW enforced security for authorized software
Background

• Approach facilitates application development and transformation for TEE using ARM TrustZone

• Automatically partitioning existing Android app.

• Unidirectional TEE execution model

• Lack of standardization -> just few Android app. use this technology
• TEE offers Trusted Applications (TAs)
  • TA composed of TEE Commands
  • Providing services to clients of the TA
  • Enforcing confidentiality, integrity and access rights for resources and data
  • Each TA is independent and protected against ecosystem of the application providers
  • TAs can access secure resources and services
    • key management
    • cryptography
    • secure storage
    • secure clock
    • trusted display
    • trusted virtual keyboard via TEE Internal API.
• Client applications running in the rich OS can access and exchange data with TAs via TEE Client API.

Figure 1: TEE system architecture
Figure 2: An overview of the approach
PHASE 1
INPUT:

+ Android App (binary)
+ Source:
  Any method that reads and returns confidential data.
+ Sink:
  Writes confidential data into a resource that can be accessed or controlled outside the application.

Gray area -> external components
PHASE 2:

- **Partitioning Framework**
  - Generates candidate code segments to be deployed as TEE commands of a TA

- **Algorithm:** Selection of candidate program segments
PHASE 3:

• Grouping statements operating on conf. data

• Including:
  • Code segments that manipulate OS-dependent code
  • Confidential operations with overlapping contexts which cannot be isolated
  • Code fragments control-dependent on conf. data
PHASE 4:

- Assisting the engineer in transforming code fragments into TEE Commands.
- Autom. generated code with TEE API calls for establishing communication and parameters passing btw normal and secure world

Gray area -> manually supported components
Partitioning Framework

• Starting with taint analysis enhanced with annotation of taint-propagating statements with contextual information

• Classifying the annotated statements and capture a subset of the statements that will form a secure partition to be deployed on TEE

• Then identifying groups of statements

• Resolve corner cases

• To maintain the flow of data through transfer statements -> substitute confidential data references with *opaque references* in the transformed application
Unique Opaque References

- Secure transfer of confidential data btw. normal world and secure world.
- 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.
- It’s 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.
- A reference is created by allocating a new unique Java object of a required type.
Unique Opaque References

• Avoiding compile and runtime errors by generating opaque references of types as expected by the original implementation.

• Uniquely identify primitive types:
  - Applying minor code refactoring on the original application
  - Substitute tainted primitive variables with objects of primitive wrapper classes.

• Opaque references do not conflict with polymorphic method invocations.
  - Polymorphic method invocations with tainted base objects are marked as privileged and deployed in TEE Commands
  - The runtime type of a base object (its opaque reference) does not affect the control flow of the application.
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 $D_m$ do
9:             $T_{stmt} \leftarrow getTage(stmt)$
10:            if isAnnotated(stmt) \& (\exists t \in T_{stmt} : D_m \rightarrow t) then
11:                \> Process tagged statement with matching method context:
12:                $OUT \leftarrow OUT \cup \{processStatement(stmt, m)\}$
13:             $M_{cache} \leftarrow M_{cache} \cup m$
14:             $M \leftarrow M \setminus m$
15:          procedure processStatement($n, m$)
16:          $P_n \leftarrow getInTaintSetOf(n, D_m)$
17:          $R_n \leftarrow getOutTaintSetOf(n, D_m)$
18:          STAGE 1: Extend the worklist
19:             \> Transfer call statement with a tainted parameter:
20:                if isCallStatement($n$) \& (params($n$) \& $P_n \neq \emptyset$) then
21:                    $M \leftarrow M \cup \{getCallee(n, G)\}$  \> add callee to the worklist
22:                    return $\emptyset$
23:             \> Returning taint – add callers of $m$ to the worklist:
24:                if isExitStatement($n$) then
25:                    $M \leftarrow M \cup \{callersOf(m, G)\}$
26:                    return $\emptyset$
27:             \> Taint flows to a field variable – add callers of class methods to the worklist:
28:                if $\exists r \in R_n \land isFieldVar(r)$ then
29:                    $c \leftarrow getDeclaringClass(r)$
30:                    for $m$ in getMethodsOf($c$) do
31:                        $M \leftarrow M \cup \{callersOf(m, G)\}$
32:                \> Source stmt taints parameters of enclosing method – add callers of $m$ to the worklist:
33:                if ($n \in S$) \& ($D_n \neq \emptyset$) then
34:                    $M \leftarrow M \cup \{callersOf(m, G)\}$
35:                STAGE 2: Record privileged statement
36:                if isPrivilegedStatement($n$) then
37:                    return ($n, R_n, P_n$)
38:                else
39:                    return $\emptyset$  \> transfer statements are not added

- **Input:**
  - List of sources
  - List of sinks
  - Interprocedural CFG (control-flow-graph)
  - Worklist of methods

- **Output:**
  - Map of candidate privileged stmts and associate in/output taint sets

- **Stage 1**
  - Extending the worklist

- **Stage 2**
  - Classifying taint-propagation stmts
Implementation

Figure 4: System implementation

General view of the components
Figure 5: Generated and transformed source code
Experimental Evaluation

• 6 real-world applications and a set of micro-benchmarks on SierraTEE

• Standard Android Benchmarks
  • -> Droidbench and SecuriBench
    • Designed to check taint analysis for different cases of data flow arising in secure context.

• -> Control-dependent
  • Text extension from the authors for extracting the decision part of the control structure as a TEE Command
• Total:
  • Number of cases of confidential data flow from source to sink
  • Each benchmark obtained through taint analysis

• Correct:
  • Prototype framework applied
  • Manually checked partition
  • Results -> number of cases where resulting transformation is successful

• -> 86% of cases were successfully partitioned and transformed.

<table>
<thead>
<tr>
<th>SecuriBench</th>
<th>Correct/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliasing</td>
<td>5/5</td>
</tr>
<tr>
<td>Arrays</td>
<td>6/6</td>
</tr>
<tr>
<td>Basic</td>
<td>30/40</td>
</tr>
<tr>
<td>Collections</td>
<td>11/11</td>
</tr>
<tr>
<td>DataStructures</td>
<td>5/5</td>
</tr>
<tr>
<td>Factories</td>
<td>3/3</td>
</tr>
<tr>
<td>Inter</td>
<td>11/12</td>
</tr>
<tr>
<td>Pred</td>
<td>6/8</td>
</tr>
<tr>
<td>StrongUpdates</td>
<td>4/4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DroidBench</th>
<th>Correct/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliasing</td>
<td>1/1</td>
</tr>
<tr>
<td>ArraysAndLists</td>
<td>2/3</td>
</tr>
<tr>
<td>FieldAndObjectSens</td>
<td>7/7</td>
</tr>
<tr>
<td>GeneralJava</td>
<td>23/23</td>
</tr>
<tr>
<td>ImplicitFlows</td>
<td>1/2</td>
</tr>
<tr>
<td>DecisionProtecSimple</td>
<td>9/12</td>
</tr>
<tr>
<td>DecisionProtec</td>
<td>6/8</td>
</tr>
</tbody>
</table>
Case Study

• 6 widely-used open-source applications
  • Google Authenticator
  • Tiqr
  • OpenKeychain
  • Card.io
  • Hash it!
  • Pixelknot
Summarize of the contribution of commands to the TCB size in SierraTEE and the change to the client code.
• It compared the TEE command with the execution time of the original Java code in Android OS but not deployed to TEE.

• Table 3 -> computation in TEE is faster than the original application.

• Not surprising -> execution in C code is usually faster than execution in Java code.

• Most of the Overhead:
  • Penalty for setting up TEE context
  • Establishing TEE session
  • Switching between normal and secure world

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