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 tranformaion for TEE using ARM TrustZone
- Automatically partitionig existing Android app.
- Unidirectional TEE execution model
- Lack of standardization -> just few Andorid 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 agianst ecosystem of the application providers
  - TAs can access secrue 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.

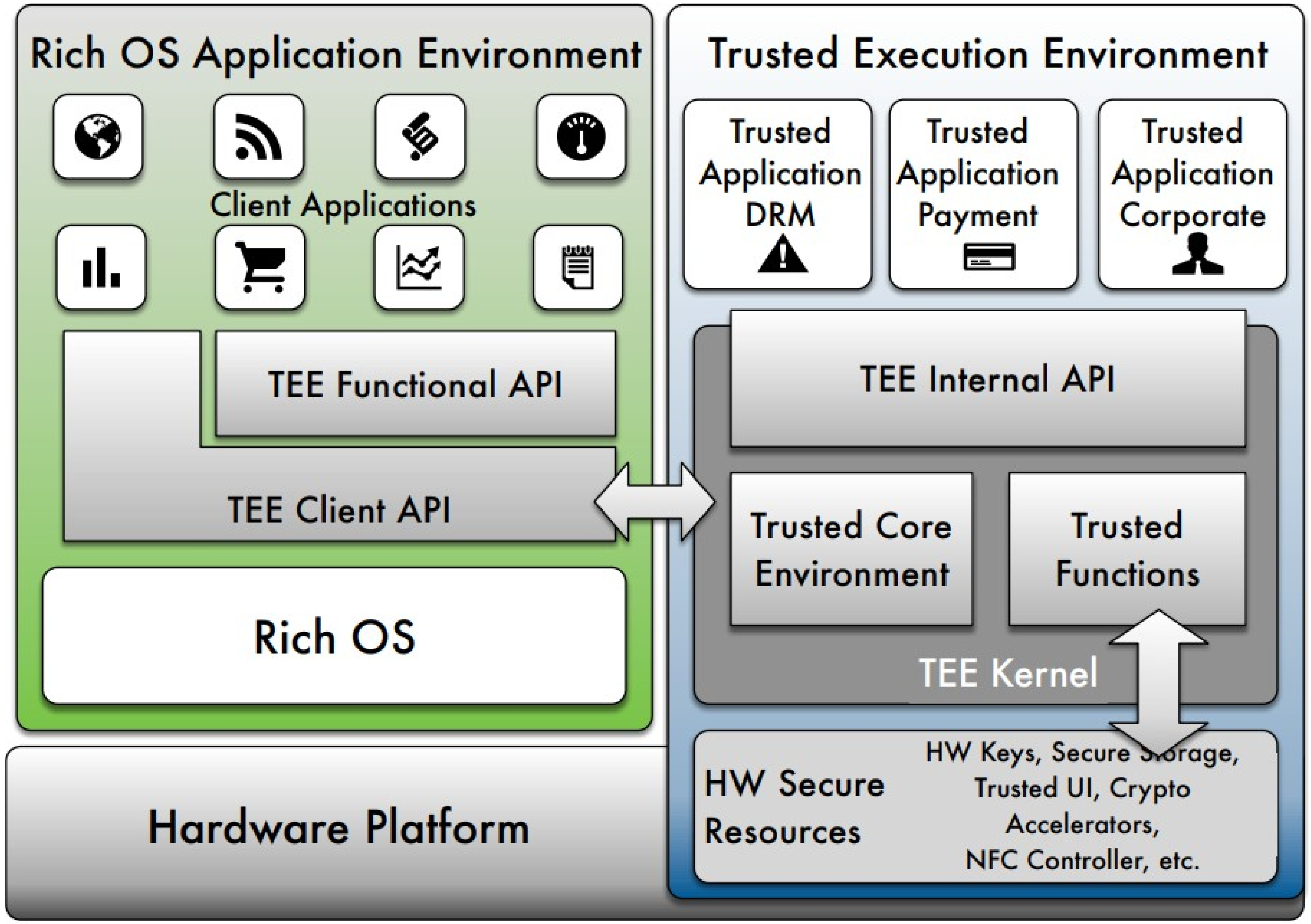

Figure 1: TEE system architecture

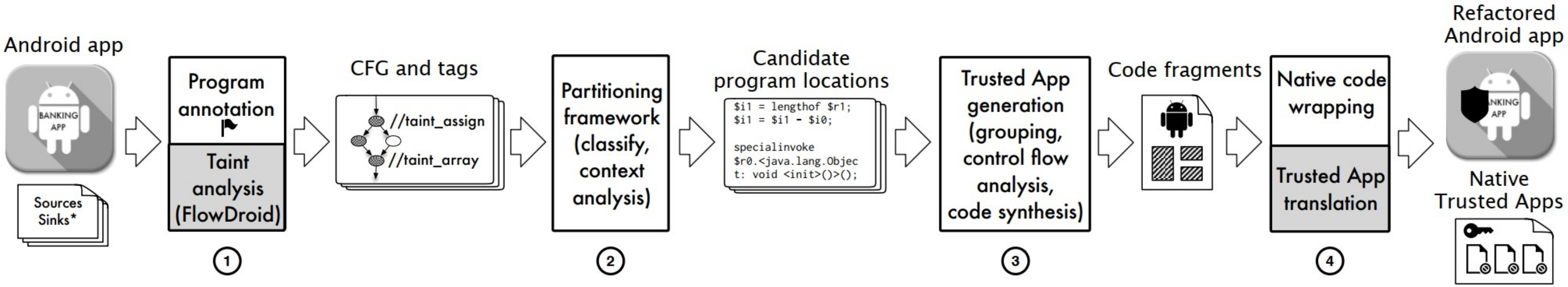

Figure 2: An overview of the approach

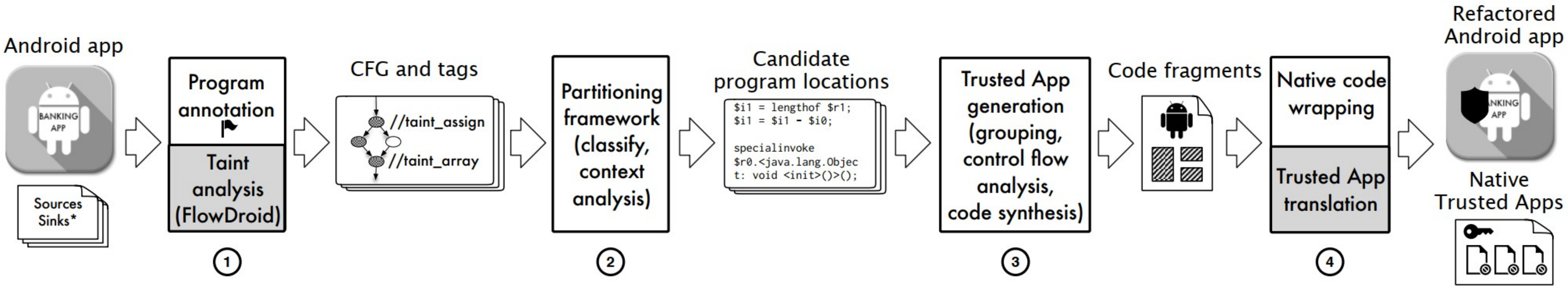

Figure 2: An overview of the approach

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

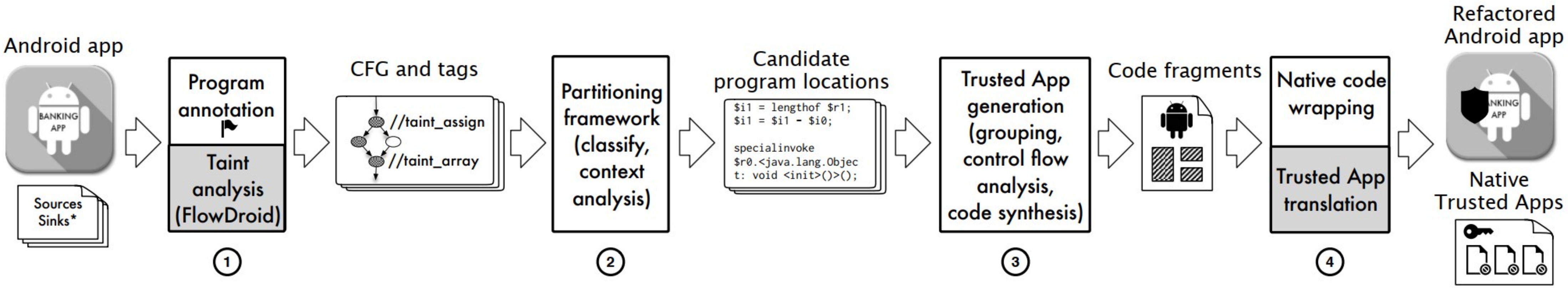

Figure 2: An overview of the approach

PHASE 2:

- Partitioning Framework
  - Generates candidate code segments to be deployed as TEE commands of a TA
- Algorithm: Selection of candidate program segments

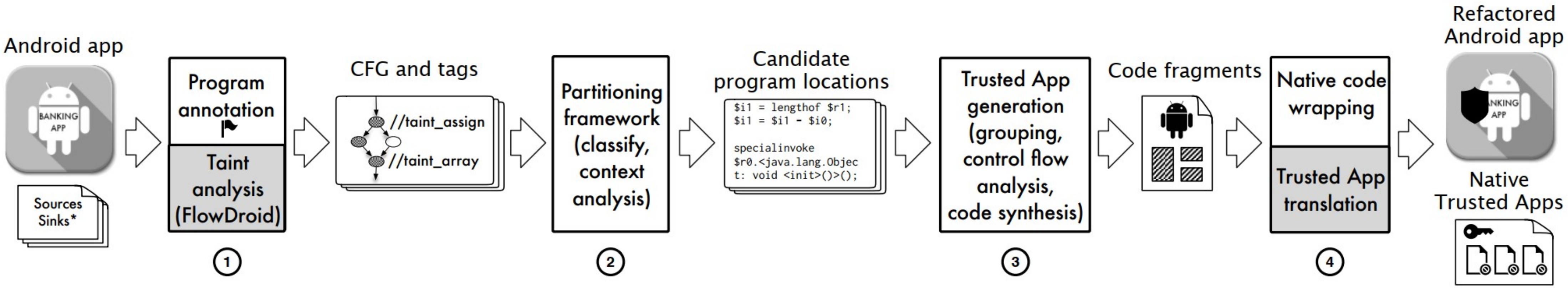

Figure 2: An overview of the approach

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

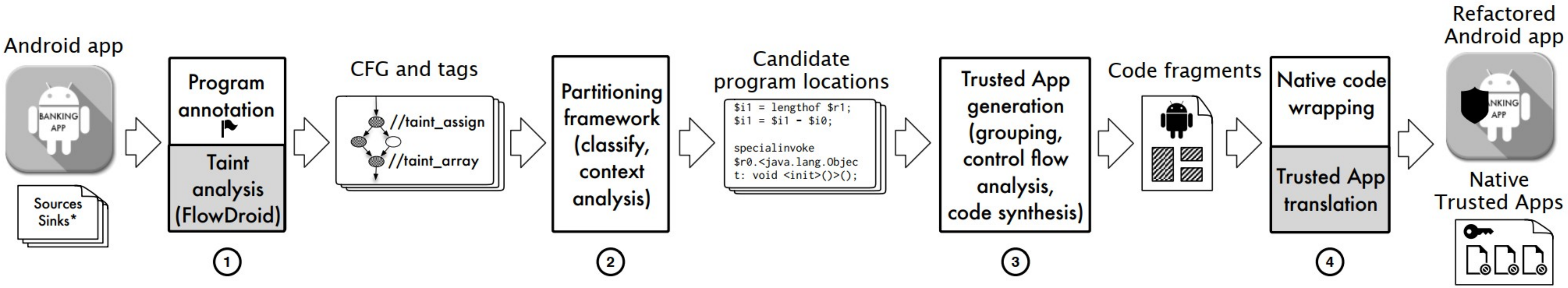

Figure 2: An overview of the approach

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 wolrd.
- 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

$M \leftarrow \emptyset$; $M_{cache} \leftarrow \emptyset$
for s in S **do**
  $M \leftarrow M \cup \{methodOf(s)\}$ ▷ Initialize worklist of methods
while $M \neq \emptyset$ **do**
  $m \leftarrow pick(M)$
  if $m \notin M_{cache}$ **then**
    $D_m \leftarrow getMethodContext(m)$
    for $stmt$ in m **do**
      $T_{stmt} \leftarrow getTags(stmt)$
      if $isAnnotated(stmt) \wedge (\exists t \in T_{stmt} : D_m \Rightarrow t)$ **then**
        ▷ Process tagged statement with matching method context:
        $OUT \leftarrow OUT \cup \{\text{processStatement}(stmt, m)\}$
    $M_{cache} \leftarrow M_{cache} \cup m$
    $M \leftarrow M \setminus m$
procedure processStatement(n, m)
  $P_n \leftarrow getInTaintSetOf(n, D_m)$
  $R_n \leftarrow getOutTaintSetOf(n, D_m)$
STAGE 1: Extend the worklist
  ▷ Transfer call statement with a tainted parameter:
  if $isCallStatement(n) \wedge (params(n) \cap P_n \neq \emptyset)$ **then**
    $M \leftarrow M \cup \{getCallee(n, G)\}$ ▷ add callee to the worklist
    return $\emptyset$
  ▷ Returning taint – add callers of m to the worklist:
  if $isExitStatement(n)$ **then**
    $M \leftarrow M \cup \{callersOf(m, G)\}$
    return $\emptyset$
  ▷ Taint flows to a field variable – add callers of class methods to the worklist:
  if $\exists r \in R_n \wedge isFieldVar(r)$ **then**
    $c \leftarrow getDeclaringClass(r)$
    for m in $getMethodsOf(c)$ **do**
      $M \leftarrow M \cup \{callersOf(m, G)\}$
  ▷ Source stmt taints parameters of enclosing method – add callers of m to the worklist:
  if $(n \in S) \wedge (D_m \neq \emptyset)$ **then**
    $M \leftarrow M \cup \{callersOf(m, G)\}$
STAGE 2: Record privileged statement
  if $isPrivilegedStatement(n)$ **then**
    return (n, R_n, P_n)
  else ▷ transfer statements are not added
    return $\emptyset$

- 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

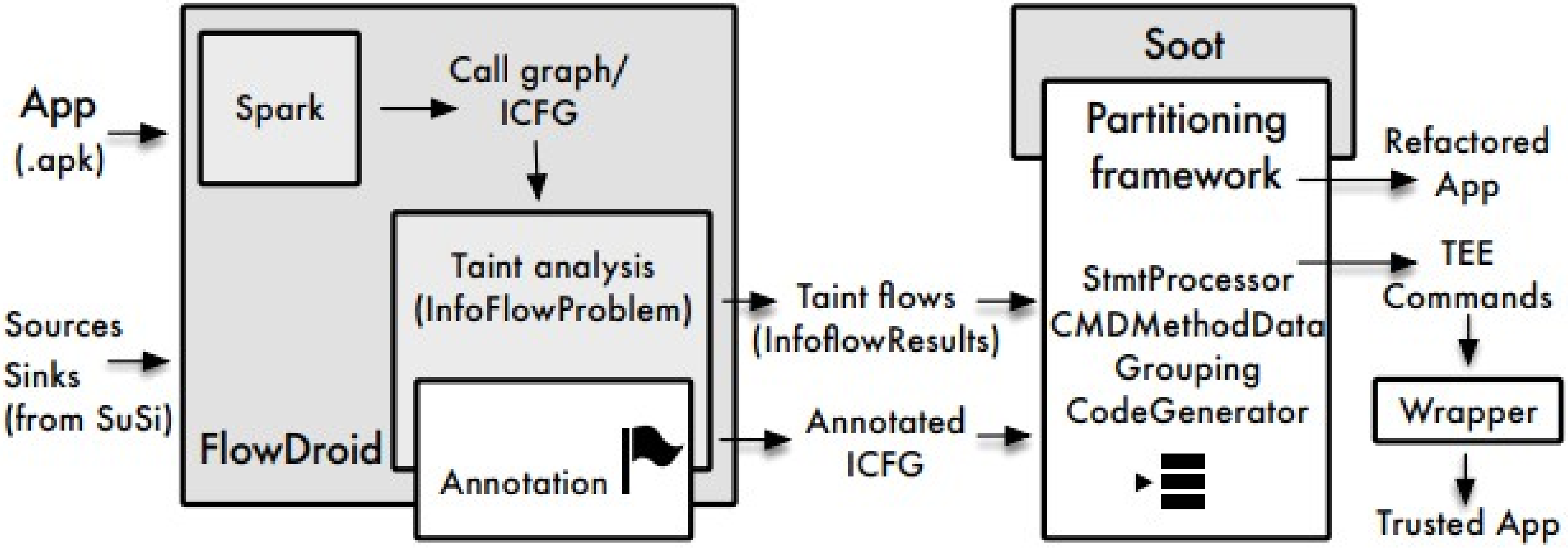

Figure 4: System implementation

General view of the components

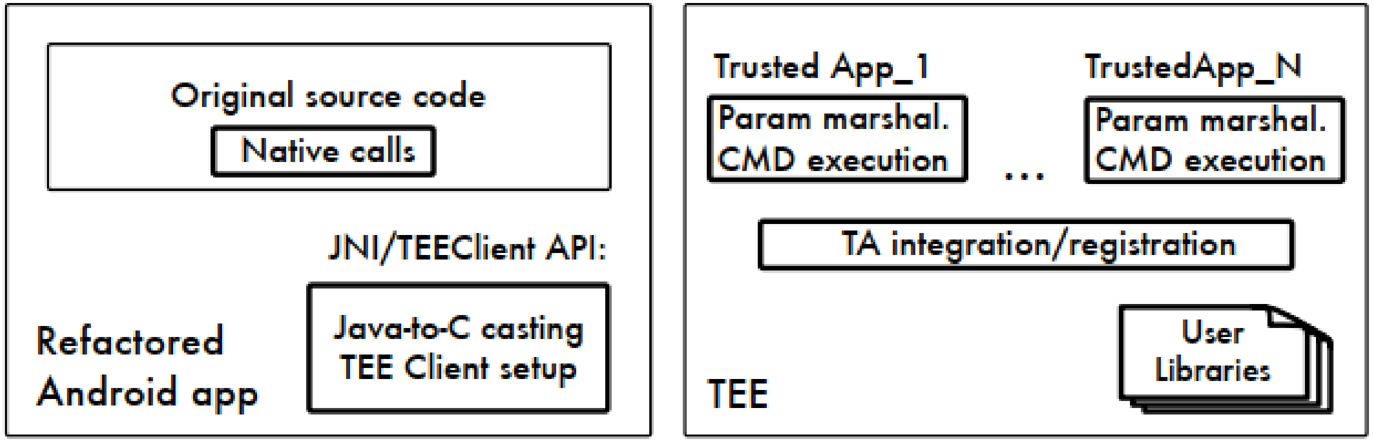

Figure 5: Generated and transformed source code

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 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		

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.

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

- 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 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 μs (0.9)	9 μs (15)	9 μs (10)
Multiply	140 μs (10)	30 μs (11)	30 μs (10)
GA TOTP	640 μs (107)	40 μs (4)	85 μs (18)
GA HOTP	600 μs (28)	40 μs (3)	70 μs (20)
tiqr CMD1	14 μs (3)	13 μs (1)	250 μs (35)
tiqr CMD2	21 μs (6)	13 μs (1)	220 μs (10)
tiqr CMD3	2.5 μs (0.4)	0.8 μs (0.04)	78 μs (5)
tiqr CMD4	19 μs (4)	10 μs (0.5)	220 μs (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 μs (0.8)	0.7 μs (0.03)	78 μs (5)
CI CMD2	3.2 μs (0.7)	0.6 μs (0.06)	79 μs (5)
PK CMD1	3.2 μs (0.5)	0.9 μs (0.06)	86 μs (6)
PK CMD2	4.6 μs (0.4)	0.7 μs (0.03)	80 μs (5)
PK CMD3	1.99 s (0.0001)	26 μs (3)	280 μs (34)
PK CMD4	2.11 s (0.0002)	27 μs (5)	267 μs (32)
Hash it!	557 μs (61)	27 μs (5)	71 μs (10)

Thank you for your attention!