and tackle unknowns. A memory object of the application is represented as a tuple $\phi(x) := \langle \phi_T(x), \phi_K(x), \phi_V(x) \rangle$. $\phi_T(x)$ specifies its type, which can be Java primitive types (e.g., int, long, char) as well as class types (e.g. String, StringBuilder). AppAudit introduces object kind $\phi_K(x)$ to distinguish known values and unknowns. $\phi_K(x)$ can be one of the following cases: 1) a *concrete object (CON)* that is created during the execution process, e.g. an object created by the `new` instruction; 2) a *prior unknown (PU)*, which exists prior to the execution process and contains no known values to the executor, e.g. a global variable; 3) a *derived unknown (DU)*, which was a prior unknown but is changed during the execution process. DUs mix known values and unknowns. For instance, a DU could have some known fields and some unknown fields. $\phi_V(x)$ stores the known value(s) of the object. For primitive types, $\phi_V(x)$ reflects its known value, e.g. an integer of value 5 is represented as $\phi_V(x) = \{val \mapsto 5\}$. If the value is unknown, $\phi_V(x) = \emptyset$. For class types, $\phi_V(x)$ stores all its known fields, e.g. $\phi_V(x) = \{field1 \mapsto \phi(y)\}$ representing $x.field1 == y$. Unknown fields will not appear. Arrays are special objects with indices as fields, e.g. an array of two elements is represented as $\phi_V(x) = \{0 \mapsto \phi(y), 1 \mapsto \phi(z), length \mapsto 2\}$. $\phi_V(x)[field]$ can query a particular filed of an object x. If the field is known, this query returns the known object. Otherwise, this expression returns a prior unknown.

In addition to our object representation, AppAudit also tracks taints on objects similar to dynamic taint analysis [11], [12]. For each memory object x, we define $\tau(x)$ as its tainting state. Each source API could generate a specific type of taint, representing a particular type of personal data (e.g. text message, location, etc). Taints propagate along with the object. Any object derived from a tainted object will also be tainted. If a sink API meets a tainted object, our executor will report a leak. We will explain our tainting rules in details after introducing the execution rules.

B. Basic Execution Flow

We use five examples to demonstrate the basic workflow of the executor and the expressiveness of our object representation. We assume that the `source()` API generates a tainted integer (denoted as *taint*) and the `sink()` API checks if its parameter is tainted. All parameters and global variables (static class fields) are prior unknowns when the execution starts, whose values are unknown to the executor.

1) In `foo1` shown below, c is first assigned a new concrete object with no known fields, i.e., $\langle T, Concrete, \emptyset \rangle$. Then $c.f$ is assigned and c becomes $\langle T, Concrete, \{f \mapsto taint\} \rangle$. Finally, `sink()` checks $c.f$. And since it is a taint, the executor reports a leak.

```
foo1(T x, T y) {
  c = new T();
  c.f = source();
  sink(c.f);
}
```

2) In `foo2` shown below, x starts as a PU. Then $x.f$ is assigned with a concrete object (the taint), which changes x from a prior unknown to a derived unknown $\langle T, DU, \{f \mapsto taint\} \rangle$. A derived unknown implies that this object was unknown (PUs) but some known values have been assigned to it during the execution. Therefore, when the concrete object c gets $x.f$, it gets the known value (the taint) assigned to $x.f$ before. Finally, the executor successfully reports the leak on `sink()`.

```
foo2(T x, T y) {
  x.f = source();
  c = new T();
  c.f = x.f;
  sink(c.f);
}
```

3) In `foo3` shown below, the condition checks if a concrete object c is equal to a prior unknown x. By definition, a prior unknown is created before execution while a concrete object is created afterwards. Thus the executor can safely evaluate the condition to be false and no leak will be reported.

```
foo3(T x, T y) {
  c = new T();
  if (c == x)
    sink(source());
}
```

4) In `foo4` shown below, the condition compares two prior unknowns. Since the executor does not know if x and y refer to the same object, this condition ends up as an unknown. The branching depends on an unknown condition and thus the executor reverts to the approximation mode, which will be discussed in details later.

```
foo4(T x, T y) {
  if (x != y)
    sink(source());
}
```

5) In `foo5` shown below, x changes to a derived unknown with a concrete field but y is still a prior unknown when its field is checked. Thus, the executor also needs to revert to approximations.

```
foo5(T x, T y) {
  x.f = source();
  sink(y.f);
}
```

From these examples, we illustrate how our object representation keeps record of both known and unknown objects and tracks their propagation to reflect the data flows of personal data.

C. Complete Execution Rules

Table II lists the complete execution rules used in AppAudit executor. Rule (1) to (7) have been covered by above-mentioned examples. The rest handle other bytecode instructions:

Function Call. Rule (8) shows that our dynamic analysis is naturally inter-procedural. When a function call is being made during execution, the executor will step into the function and pass the parameters accordingly.

Arithmetic Operations. Rule (9) and rule (10) outline how to evaluate binary and unary arithmetic expressions. These expressions take only primitive types. Basically the executor will compute concrete values when both operands have concrete values. When unknowns are present, the result will be unknown accordingly.

Comparison Operations. Rule (11) tackles comparison expressions. Similar to arithmetic operations, if both operands are concrete (known), the result can be evaluated naturally. When unknowns are present, the result is also unknown except for one case. If one operand is a concrete object while the other is an unknown (PU or DU), the result is definitely evaluated to false.

Array Operations. Rule (12) and rule (13) handle array operations, which are similar to rule (4) and rule (5). Changing an array element can also change a prior unknown to a derived unknown.

Branching Operations. Rule (14) and rule (15) handle branching instructions. For conditional jumps with unknown conditions, the executor will revert to the approximation mode.

D. Tainting Rules

Personal data is marked as *tainted*, which is propagated during execution. The taint tracking capability of AppAudit is largely similar to the taint propagation rules used in dynamic taint analysis [11], [8]. In our rules, rule (1) and rule (6) set taints explicitly. Rule (9) and rule (10) taint the result as long as one of its operands is tainted. Rule (12) taints x as long as i is tainted. This rule handles encryption libraries that perform substitution to encrypt data, hence tainted inputs should lead to tainted outputs.

E. Execution Extensions and Optimizations

Our executor contains several extensions and optimizations to accelerate the execution speed while maintaining the same instruction semantic.

Dynamic Dispatch (Reflection and Virtual Calls). Java virtual calls and reflections are dynamically dispatched. During execution, these call targets will be resolved.

Inlining Call-back Functions. As mentioned before, call-back functions are widely used and hide implicit data flows. Thus, when the executor encounters a trigger API, it will execute the callback function being registered after the current function is finished.

Exception Control Flow. Exceptions can affect control flows. Some instructions and APIs can generate exceptions (e.g. array indexing instructions and file related APIs). Currently our executor supports only plain exceptions (no nested exceptions). Unhandled exceptions are ignored during execution.

Library Emulation. Library functions contain large body of instructions and lots of calls into other library functions. The executor emulates some library functions for improved analysis performance. To emulate a particular library function, the executor manipulates its object representations directly to achieve the same effect of the emulated function. For example, to emulate `swap(x,y)`, the executor swaps the object representation directly without executing its bytecodes. Library emulation is commonly implemented to accelerate the calls to standard Java library functions.

Infinity Avoidance. During execution (both the *exec* and *approx* mode), our executor can run into infinity due to infinite loops and recursions in the application. For example, the application can spawn a thread that uses an infinite loop to check network updates. In real execution, this thread could be interrupted by user exiting the application. However, in approximated execution, this infinite loop will never end. To ensure that our analysis can always terminate, we design a threshold-based approximation to detect and terminate infinities. Both avoidance mechanisms lead the executor to the *leap* state when infinite loops or recursions are detected.

We introduce a counter to record how many instructions have been executed for a particular function. If this counter exceeds the total instruction count of the function times a certain threshold, we will cut short the execution.

Similarly for infinite recursions, we monitor the call stack during execution. If the depth exceeds a designated threshold, the executor assumes a stack overflow happens and then terminates the execution.

We obtain these two thresholds through empirical experiments. First we turn on instruction tracing such that every instruction being executed by AppAudit will be logged. Then we gradually increase the threshold until the infinity avoidance mechanism no longer cuts short any code paths. Finally, we double this fix point as our final threshold to ensure that these thresholds work for other real apps.

F. Approximation Mode

As shown in the execution rules, unknown values can be stored, propagated and evaluated with our object model. However, when a conditional jump instruction meets unknown values, the executor will fail to perform control flow

#	Instruction	Execution Semantic
(1)[1]	`x=12`	$\phi(x) \leftarrow \langle int, \mathbf{CON}, \{val \mapsto 12\}\rangle$
(2)[2]	`x=new T()`	$\phi(x) \leftarrow \langle T, \mathbf{CON}, \emptyset\rangle$
(3)[12]	`x=y`	$\phi(x) \leftarrow \phi(y)$
(4)[2]	`x.f=y`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(x), \mathbf{DU}, \phi_V(x) \cup \{f \mapsto \phi(y)\}\rangle, & \text{if } \phi_K(x) = \mathbf{PU}\\ \phi_V(x) \leftarrow \phi_V(x) \cup \{f \mapsto \phi(y)\}, & \text{otherwise}\end{cases}$
(5)[2]	`x=y.f`	$\phi(x) \leftarrow \phi_V(y)[f]$
(6)[1]	`x=source()`	$\phi(x) \leftarrow taint$
(7)[1]	`sink(x)`	switch to check mode
(8)[12]	`call fn(`$e_0, \cdots$`)`	assign parameters according to Rule (3)
(9)[1]	`x=y binop z`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(y), \mathbf{CON}, \kappa_{binop}(\phi_V(y), \phi_V(z))\rangle & \text{if } \phi_K(y) = \mathbf{CON} \wedge \phi_K(z) = \mathbf{CON}\\ \phi(x) \leftarrow \langle \phi_T(y), \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(10)[1]	`x=unop y`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(y), \mathbf{CON}, \kappa_{unop}(\phi_V(y))\rangle & \text{if } \phi_K(y) = \mathbf{CON}\\ \phi(x) \leftarrow \langle \phi_T(y), \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(11)[12]	`x=y cmp-op z`	$\begin{cases}\phi(x) \leftarrow \langle Bool, \mathbf{CON}, \kappa_{cmp}(\phi_V(y), \phi_V(z))\rangle & \text{if } \phi_K(y) = \phi_K(z) = \mathbf{CON}\\ \phi(x) \leftarrow \langle Bool, \mathbf{CON}, \{val \mapsto false\}\rangle & \text{if } \phi_K(y) \neq \phi_K(z) \wedge \phi_T(y) \in PRIMITIVE\_TYPES\\ \phi(x) \leftarrow \langle Bool, \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(12)[12]	`x=a[i]`	$\begin{cases}\phi(x) \leftarrow \phi_V(a)[\phi_V(i)[val]] & \text{if } \phi_K(a) = \mathbf{CON} \wedge \phi_K(i) = \mathbf{CON}\\ \phi(x) \leftarrow \langle ELEM_T, \mathbf{DU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(13)[12]	`a[i]=x`	$\begin{cases}\phi_V(a) \leftarrow \phi_V(a) \cup \{\phi_V(i)[val] \mapsto \phi(x)\} & \text{if } \phi_K(a) = \phi_K(i) = \mathbf{CON}\\ \phi(a) \leftarrow \langle \phi_T(a), \mathbf{DU}, \phi_V(a) \cup \{\phi_V(i)[val] \mapsto \phi(x)\}\rangle & \text{if } \phi_K(a) \neq \mathbf{CON} \wedge \phi_K(i) = \mathbf{CON}\end{cases}$
(14)	`jmp-op cond,l`	$\begin{cases}pc \leftarrow \kappa_{jmpop}(\phi_V(cond), pc, l) & \text{if } \phi_K(cond) = \mathbf{CON}\\ \text{switch to approx mode} & \text{otherwise}\end{cases}$
(15)	`jmp l`	$pc \leftarrow l$

[1] this bytecode accepts primitive types
[2] this bytecode accepts class types

Table II: The execution rules. κ is a series of evaluation functions that perform real calculation when values are known.

decision. In this case, the executor changes to approximation mode.

Unknown Branching Approximation. The executor relies on this approximation to continue when it encounters branching instructions with unknown conditions. This approximation is designed to skip unknown loops, since these loops cannot provide useful known information from unknowns. Table III shows the four basic control flow structures compiled by an Android compiler. For the three looping structures, the branching approximation always chooses not to take the conditional branch to skip these loops. However, as we cannot distinguish ifs and loops, this approximation will only explore the "then" branch for unknown if-else structures. This bias is benign. Consider the following program:

```
foo() {
  T a = new T();
  T b = new T();
  bar(a,a);
  bar(a,b);
}
bar(T x, T y) {
  if (x == y)
    return ;
  else
    sink(source());
}
```

In this example, `bar` will be executed. When executing `bar`, both `x` and `y` are prior unknowns, which trigger the approximation to guide the executor to explore only the "then" branch and thus no leak will be reported. Due to insufficient calling contexts, the "else" branch will not be explored when analyzing `bar`.

Then according to our API analysis, `foo` will also be analyzed if `bar` has been analyzed, as `foo` is a caller of `bar`. When analyzing `foo`, the executor will analyze `bar` again with two concrete calling contexts. Under the `bar(a,a)` context, the condition will be evaluated to true and no leak will be reported. Under the `bar(a,b)` context, the condition will be evaluated to false and the leaking "else" branch will be explored.

Observed from this case, the unknown branching approximation only affects a function with insufficient calling contexts (`bar`). The approximation will result in fewer code paths being explored. But then the executor will reach callers (`foo`) of this function and re-analyze the unsuccessful function (`bar`) with more concrete calling contexts from its caller. If the program contains leaking paths, then at least one of these calling contexts will be sufficient to reach the leaking point. Thus the bias introduced before will be amortized. If the program does not contain leaking paths, then the approximation will skip some code paths but none

```
if (<cond>) { <then> }
else { <else> }
<rest>
```

(a) if statement

```
for (<init>;
    <cond>;
    <incr>) {
  <body>
}
<rest>
```

(b) for loop

```
while (<cond>) {
  <body>
}
<rest>
```

(c) while loop

```
do {
  <body>
} while (<cond>);
<rest>
```

(d) do-while loop

(a) if statement

```
cond_label:
▷ jump, !<cond>, else
  <then>
  goto rest
  <else>
  goto rest
  <rest>
```

(b) for loop

```
  <init>
cond_label:
▷ jump, <cond>, body
  <rest>
  <body>
  <incr>
  goto cond_label
```

(c) while loop

```
cond_label:
▷ jump, <cond>, body
  <rest>
  <body>
  goto cond_label
```

(d) do-while loop

```
  <body>
cond_label:
▷ jump, <cond>, body
  <rest>
```

Table III: Four basic control flow structures and their compiled bytecode streams.

of them will leak data. In short, the unknown branching approximation will not miss leaking code paths.

G. Accuracy Analysis

Since our executor performs dynamic analysis, we would like to ensure that it does not miss important (leaking) code paths. When running in execution mode, the executor can faithfully reproduce the actual path of the real execution. When the executor turns to the approximation mode, it will explore only a few possible code paths, which could lead to false negatives. We have analyzed the side-effect of unknown branching approximation. When the application contains leaking paths, the executor will have a proper calling context for executing regardless of this approximation. Thus this approximation only misses non-leaking paths and is benign to the overall accuracy. Our infinity avoidance relies on two thresholds to cut short infinite loops and recursions. Both are obtained from empirical experiments.

In addition, our executor embodies a taint analysis to track the dissemination of personal data. Thus the correctness of taint rules also affects the accuracy of the executor. We identify the following cases that could affect accuracy of a dynamic taint analysis:

Taint Sanitization. Currently, our tainting rules only add and propagate taints but never remove them. This could lead to inaccuracy and false positives. One typical case is about rule (9) and rule (10) in Table II. Currently, the result of an arithmetic operation will be tainted as long as one operand is tainted. However some arithmetic operations always return the same result regardless of the value of the operands. For example, $x = y \oplus y$ always returns zero. For such cases, the taint on the result should be removed. Our current implementation does not have taint sanitization. Nevertheless, although these cases are possible for hand-crafted applications, the standard Android Java compiler never generates such idioms.

Array Indexing. For an array operation $x = a[i]$, currently x will be tainted if i is tainted. This is because encryption functions usually use an array to map plain-text inputs to encrypted outputs. Thus the taints on the output is dependent on the input. This is commonly employed by other taint analyses [11], [8] to deal with encryption libraries. However, if the array is zero-valued, then x will always be zero regardless of the index i. Again, the current propagation rule over-taints the results and could lead to false positives.

Control Flow Dependent Taints. This is a well acknowledged drawback in most taint analysis [11], [8], [13].

```
if (x == 1) y = 1;
else if (x == 2) y = 2;
```

In this case, the values of the two variables are correlated so should be their taintness. However, by using the control flow structures, the executor is unaware of the correlation and always produces an untainted y. ScrubDroid [13], [14] presents more attacking cases for a standard taint analysis. We expect to integrate a more powerful code structure recognition module to detect such cases in the future.

V. Evaluation

In this section, we evaluate AppAudit in terms of its accuracy and usability. We demonstrate the three use cases of AppAudit, with regards to market operators, app developers and mobile users. We also present a characterization study about data-leaking apps, providing guidance for designing effective data leak prevention tools.

A. Implementation

The AppAudit prototype is implemented with Java, and reflectively loads Android SDK for API signatures. Table IV presents the breakdown of source lines of code for different components. We leverage dex2jar for disassembling [15] and APKParser [16] for manifest parsing. The API analysis accounts for a relatively small portion of the entire code