Driving Execution of Target Paths in Android Applications with (a) CAR

Michelle Y. Wong
University of Toronto
Dept. of Electrical and Computer Engineering
Toronto, Canada

David Lie
University of Toronto
Dept. of Electrical and Computer Engineering
Toronto, Canada

ABSTRACT

Dynamic program analysis is commonly used to vet Android applications. One approach is targeted execution, in which interesting or suspicious code is specifically targeted and analyzed dynamically. However, faithful execution to just the paths that reach these targets can be difficult due to the dependencies they have on other parts of the application. Prior works that handle dependencies must favor either soundness or completeness to the detriment of the other. Techniques that rely on precise dependency tracking ultimately result in lower coverage of targets due to overhead. Meanwhile, other techniques that aim for completeness by ignoring or bypassing dependencies lead to unsound execution and false positives. In this paper, we treat dependencies through the lens of a path context, which represents the program state expected by the path as it is executing. We propose an approach that provides better completeness and low false positives using Context Approximation and Refinement (Car), which combines static constraint analysis and dynamic error recovery to infer a context based on the desired path flow and refine it during execution. We show that the integration of Car with targeted execution can reach 3.1× more target locations in popular Android applications than the existing state of the art while having a false detection rate of 9%, enabling more complete analysis and detection of security-sensitive behaviors.

CCS CONCEPTS

• **Security and privacy** → *Software reverse engineering*; Software security engineering; **Malware and its mitigation**.

KEYWORDS

program analysis; static analysis; dynamic analysis; malware detection; android malware; android security; mobile security; computer security; symbolic execution

ACM Reference Format:
Michelle Y. Wong and David Lie. 2022. Driving Execution of Target Paths in Android Applications with (a) CAR. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3497765

1 INTRODUCTION

Mobile devices have become an intrinsic part of daily life and the use of third-party applications on these devices provides a variety of beneficial functionality and services. With an estimated 2.8 million applications on the Google Play store, attackers who create and distribute malicious applications (i.e. malware) for gain are naturally drawn to such a large market and user base. To maintain the security of their users, application marketplaces endeavor to remove malware from their offerings by analyzing submissions to detect whether they perform any malicious behavior. Dynamic program analysis techniques are commonly used to perform this security analysis due to their precision, but they are limited by code coverage since only code that is executed during testing can be analyzed.

A challenge to these analysis is scalability. Purely dynamic analyses are generally unable to obtain significant code coverage on real applications and may thus miss malicious functionality [2, 15, 20, 27, 41]. An alternative approach is to guide the dynamic analysis with statically extracted information, so as to only execute sections of the application that are likely to contain malicious code [7, 48, 49]. Critical to the success of such approaches is the precision and scalability of the static analysis component—an overly precise static analysis will not scale, while an overly imprecise analysis may not provide the information required for the dynamic analysis to reach and execute malicious code, thus preventing it from detecting the malicious code. In particular, paths in Android applications may depend not only on the inputs to the entry point method, but on the properties of the method's parent object, as well as the properties of other objects in the application.

We represent a path's dependencies through its *context*, which we define as the constrained inputs and program state that satisfy these dependencies and are required for the path to execute. However, statically extracting the complete context does not scale [48, 49], resulting in an incomplete context and the inability to execute and dynamically analyze application paths. Guided symbolic execution [5, 30, 53] can resolve path dependencies as they arise but requires expensive symbolic tracking of all program state, which can also result in low coverage due to scalability issues, and they cannot easily integrate with dynamic analysis tools that operate on concrete execution. Other work that ignore context altogether by skipping the parts of the path that enforce the dependencies (e.g. through instrumented forced branching [34, 46] or arbitrary invocation [32]) can lead to the execution of unsound (i.e. infeasible) paths, resulting in many false positives.

In this work, we propose a different approach, **C**ontext **A**pproximation and **R**efinement (Car), to achieve a balance between forced execution, which produces unsound paths, and complete context

```java
class EnterKeyListener implements View.OnKeyListener {
  @Override
  public void onKey(View v, int code, KeyEvent event) {
    if (code == KeyEvent.KEYCODE_ENTER) {
      handleEnterKey(v, event);
  } }
  private void handleEnterKey(View v, KeyEvent event) {
    UserInputText textView = (UserInputText)v;
    textView.checkText();
} }
class UserInputText extends EditText {
  static public String keyword = null;
  int detectionMode = 0;
  public List<String> detectedInput = null;
  ...
  public void checkText() {
    if (detectionMode == 2 && detectMode2()) {
      recordInput(getText());
      <target sensitive action>
    } else { ... }
  }
  private boolean detectMode2() {
    return hasKeyword() && keyword.equals(getText());
  }
  private boolean hasKeyword() {
    return keyword != null && !keyword.isEmpty();
  }
  private void recordInput(String text) {
    detectedInput.add(text);
    write(<output file>, detectedInput);
} }
```

(a) Target path that performs a sensitive action when a keyword is detected in the user's input after they press the "enter" key

```java
/* Secondary activity in the application */
class InnerActivity extends Activity {
  @Override
  public void onResume(Bundle savedInstanceState) {
    View view = new UserInputText(...);
    View.OnKeyListener listen = new EnterKeyListener();
    process(view, listen);

    view = new OtherTextView(...);
    listen = new OtherKeyListener();
    process(view, listen);
    ...
  }

  private void process(View v, View.OnKeyListener k) {
    /* Register a key event handler for the view */
    v.setOnKeyListener(k)};
  }

  ...

  /* Called by an SMS receiver elsewhere (not shown) */
  public void onSmsReceived(SmsMessage msg) {
    if (msg.getOriginatingAddress().equals(
        ServerInfo.getAddress())) {
      UserInputText view = findViewById(...);
      view.detectionMode = extractMode(msg);
      view.detectedInput = new ArrayList<String>();
      UserInputText.keyword = extractKeyword(msg);
  } }
}
```

(b) Dependent code paths that register the key event handler with the framework and set the heap variables to their expected state

Figure 1: Example of targeting sensitive behavior in an Android application

extraction, which does not scale. Rather than tracking dependent paths precisely or ignoring them altogether, we show how the combination of static constraint analysis and dynamic error recovery can approximate a context for a path such that execution is driven to the target code location. The approximation is constructed to reduce the amount of false positives, which we define as the execution of unsound paths. Car achieves this by: (1) generating an initial context inferred from the desired control flow and (2) dynamically refining the incomplete context when any unresolved dependencies trigger errors during the path's execution. The hybrid approach to dependency resolution enables Car address the trade-offs that must be made in static dependency tracking between precision, completeness, and scalability. Through the combination of static and dynamic resolution, we can achieve much greater coverage of target behaviors in an application while maintaining reasonable soundness in the paths that are executed.

We implement Car on an existing Android targeted execution framework, Tiro [49], which statically extracts target paths for a configurable set of behaviors. The paths are then executed dynamically to trigger and analyze the behaviors. We show that the use of approximated contexts for targeted dynamic analysis enables much higher coverage of target code locations in applications, including accesses to sensitive device functionality for security analysis.

We make four main contributions:

(1) We describe Car, which effectively resolves dependencies for an targeted code path by approximating and refining its expected context through hybrid static and dynamic analysis.

(2) We design and implement Car for Android application analysis to show how context approximation can be used for targeted execution.

(3) We evaluate Car on the most popular applications in Google Play and show that it is able to reach 3.1× more non-trivial target locations for an application than the existing state of the art, with a false detection rate of 9.0%.

(4) We show that the use of approximated contexts for path driving can uncover a variety of security-sensitive behaviors in both benign and malicious applications.

We begin by providing background on the challenges of isolated path execution in Android in Section 2. We then present our design for Car in Section 3 and provide implementation details in Section 4. We evaluate Car on popular applications from Google Play in Section 5 and compare it against several state-of-the-art dynamic tools. We discuss our limitations in Section 6 and the related work in Section 7. Finally, we conclude in Section 8.

2 MOTIVATING EXAMPLE

In Figure 1, we present an example of a code path to a sensitive behavior in an Android application, in which the execution of the path depends on data from other parts of the application. While this is not extracted from a real application, it contains code patterns similar to what we have seen.

Figure 1(a) shows a sensitive action taken when a keyword is detected within the user's text input after the user presses the "enter" button. The key event handler (`EnterKeyListener`) is the entry-point of this path. It first checks if the key pressed is the "enter" key and if so, it calls `UserInputText`, a custom UI text element, to check the user's input text. It determines whether a keyword has been loaded, checks it against the user's input, and saves the input text if the keyword was detected. Finally, it performs a sensitive action if these conditions have been met (e.g. it may send the input text to a network server or access a device sensor).

If we wished to target the sensitive action and trigger it dynamically, the call path that reaches the target is:

1) `EnterKeyListener::onKey()`
2) `EnterKeyListener::handleEnterKey()`
3) `UserInputText::checkText()`
4) `<target sensitive action>`

Based on the path constraints, we can surmise that the path depends on its inputs, heap state, and Android framework state.

Figure 1(b) shows how the non-input dependencies are normally satisfied by the execution of other application paths. For example, `onSmsReceived` should be executed (with appropriate context) before `checkText` in the target path is executed. Furthermore, there may be recursive constraints or dependencies in the dependent paths; for instance, the `detectionMode` and `keyword` fields are set on receipt of an SMS message from a certain address, which might be set in yet another dependent path. Likewise, the enclosing component, `InnerActivity`, is not the application's main activity and may require navigation through multiple screens in the UI flow before it can be started.

To execute the path in Figure 1(a), the path's constraints must be resolved. A dynamic approach, such as guided symbolic execution [5, 30, 53], would resolve dependencies along the target path by symbolically tracking the input and non-input variables that are accessed (e.g. `detectionMode` and `keyword`). However, this tracking is expensive and requires a symbolic execution environment that does not integrate easily with the numerous dynamic analysis tools are based on concrete execution [11, 14, 19, 42, 43, 51, 56].

A hybrid approach could instead use static analysis to resolve dependencies and guide concrete execution of the path [34, 35, 46, 48, 49]. There are several methods of static guiding, which are differentiated by the execution abstraction used. A holistic abstraction that is faithful to normal execution (i.e. sound) is to abstract at the level of the Android framework, which manages the application's execution and facilitates access to underlying hardware. IntelliDroid [48] and Tiro [49] guides execution by injecting events into the framework to trigger a target path. They use static dependency tracking to determine exactly which framework events to inject such that the path's constraints and dependencies are resolved. For Figure 1, they might execute the following ordered chain of events to reach the target code:

i) UI or lifecycle event(s) to start `InnerActivity`
ii) Lifecycle event to register `EnterKeyListener`
iii) SMS event with an input message that sets the required values for `detectionMode` and `keyword`
iv) UI key event for the target path from `onKey()`

There are two drawbacks to this approach: (1) the breadth of the abstraction can be large and satisfying each dependency at runtime requires custom Android framework modifications for each event type; and (2) the complexity of the abstraction can become excessive for long paths, requiring the tracking of a large number and variety of constraints, which may require models for complex operations (i.e. string operation) and result in exponentially long constraint solving time. Decreasing breadth while maintaining complexity maintains precision, but limits the types of paths that can be analyzed dynamically. For instance, IntelliDroid does not support the injection of UI events, which precludes it from triggering paths in most Android applications (it would also be unable to inject the UI key event in our example). In contrast, reducing complexity by discarding paths with unsupported operations or excessive length reduces completeness.

One solution might be to decrease the amount of dependency tracking required, enabling greater coverage of target code in execution (i.e. completeness). GroddDroid [1], Harvester [34], DirectDroid [46], and Ares [6] operate on path slices leading to target code locations (i.e. they abstract at the instruction level). To guide execution, they use static instrumentation to force specific branch outcomes. Instead of resolving dependencies, this essentially bypasses them since the branch conditions are no longer enforced. Forcing branch outcomes and ignoring their dependencies can ensure that target paths are executed in full; however, this can easily lead to inconsistent data values such as the forcing of the null check in Line 26. This leads to an inconsistency if the checked object is actually null, the branch is forced anyway, and the object is then dereferenced later in the line. Forcing of multiple branches can also lead to a combination of infeasible branch outcomes, as program logic is modified without reconciling the control flow with the data compared within the branches. This can lead to the execution of unrealistic or unsound paths. As such, forced execution is well-suited for obtaining coverage but less so for the analysis of security-related behaviors, which usually require tracking how applications access and manipulate system resources and data. Due to soundness issues, it is not a reliable means for determining if an application will perform a malicious action.

Other work, such as FuzzDroid [35] or CrashScope [28], abstracts at the application level. FuzzDroid achieves semi-targeted execution by manipulating the inputs received at application entry-points and framework APIs. However, it does not track or resolve dependencies on state within the application, such as the constraints on the `detectionMode` and `keyword` fields; it would have to execute the dependent paths in order by chance to reach the target code location. This is also true for any other untargeted fuzzing tool.

In Car, we propose a fully targeted approach that achieves a balance between forced branching, which is unsound, and full dependency tracking across the Android system, which is unscalable. Instead, we target paths by abstracting at the level of code objects and guide execution by controlling the inputs, fields, and method return values accessed by a path. There are several advantages to an object-level abstraction. First, abstracting at the granularity of an object, as opposed to the entire Android framework, reduces dependencies on dynamic components like the underlying Android system, and allows injection of inputs and dependencies simply by calling object methods or setting object properties without having to understand Android framework APIs. Dependencies on state within the application and from the framework, such as heap variables or API calls, are implicitly handled, as accesses to this state are performed through field accesses and method invocations. Second, unlike smaller abstractions such as a path slice, injecting and manipulating objects enables the full execution of methods in the path, resulting in greater soundness. Also, the tracking of dependencies on object accesses ensures that the data accessed within a path is consistent. In our example from Figure 1(a), Car would trigger the target path by injecting the input object `v` such that the constrained field access to `v.detectionMode` is satisfied. It will also ensure that the static field access to `UserInputText::keyword` is satisfied by controlling the field value visible to the target path.

3 DESIGN

Using the object-level abstraction for targeted execution, Car constructs a context for a target path to resolve its dependencies on inputs, fields, and methods. However, using static dependency tracking to construct this context a priori can fail due to the trade-off between precision, completeness, and scalability. When this happens, the unresolved dependencies limit dynamic target coverage or lead to unsound paths. Car addresses this in a hybrid approach. We start from a set of statically extracted constraints that are necessarily incomplete due to the trade-offs that must be made (Section 3.1). We account for this by separating the constraint extraction and dependency resolution of a path into three levels of progressive approximation (realized, modeled, and unconstrained). Static constraints are used to infer an initial approximate context for the path. This context is constructed at run-time using a statically generated path driving framework (Section 3.2). We then refine this context dynamically by monitoring for unresolved dependencies and resolving them by re-using objects from the application's execution (Section 3.3).

3.1 Static constraint analysis

The initial static analysis to extract target paths uses an existing targeted execution framework for Android, Tiro [49]. It first constructs a conservative context-insensitive call-graph to find code paths to a set of configurable target behaviors. We define a target code path as the interprocedural control-flow path from an entry-point to a target location, where an entry-point is any location where normal execution can be transferred from Android framework code to application code (e.g. a callback method). For each path, a context- and flow-sensitive analysis extracts its constraints on inputs, fields, and methods that define its execution. Constraints on inputs can be resolved by injecting the path's entry event with values derived from the solved constraints. Constraints that cannot be resolved through entry-point inputs (e.g. heap or framework API values) form dependencies on other paths in the application.

The separation of the initial path extraction from the constraint analysis balances the need for coverage when detecting target behaviors and the need for precision when extracting path constraints. The precision required to determine the exact program values required for the path (akin to symbolic execution) is expensive and impractical to perform over the entire application. The trade-off between the precision and cost of this analysis impacts its completeness: to fully analyze all constraints, it requires performing precise symbolic constraint analysis over methods directly in the target's call path and all side/auxiliary methods they invoke; essentially, this is comprises of the sub-graph of the call-graph starting at the path's entry-point, which can be prohibitively large. In our earlier example from Figure 1, this would require constraint analysis over the methods in the target call path as well as the auxiliary methods `detectMode2()`, `hasKeyword()`, `getText()`, `recordInput()`, `write()`, and any other methods they may invoke.

A purely static analysis can sacrifice the precision of this analysis for scalability such that constraints and dependencies can be tracked across this sub-graph and across the application (albeit imprecisely). However, in a hybrid system where the static results are then used to generate values for execution, this can result in under-constrained program state that does not actually resolve a path's dependencies dynamically, resulting in the incomplete execution of the path. For instance, the imprecise results of an any-path analysis may indicate that a constrained variable may have one of several values (due to the union of data flows from different paths) while only one of these values is actually valid for a particular target path. Previous hybrid tools accounted for this by using a different trade-off where precision is maintained but completeness is heuristically scaled back: IntelliDroid [48] and Tiro [49] only perform constraint analysis on the methods directly in the target's call path and one level of auxiliary methods they invoke, and Harvester [34] uses a cut-off value to limit the depth of analysis into callers and callees when computing the path slice to force execute. Constraints imposed by code in deeper auxiliary/callee methods are ignored and unresolved. This is propagated when the constraints are used to perform dependency tracking, resulting in unresolved dependencies and incomplete execution of target paths.

The key intuition behind Car's design is that a combination of both static and dynamic techniques can be used to mitigate this limitation. We directly address the trade-off between precision, completeness, and scalability by separating the dependency resolution of a target path into three levels of approximation, each of which are handled separately. This separation is illustrated in Figure 2 for the code example in Figure 1.

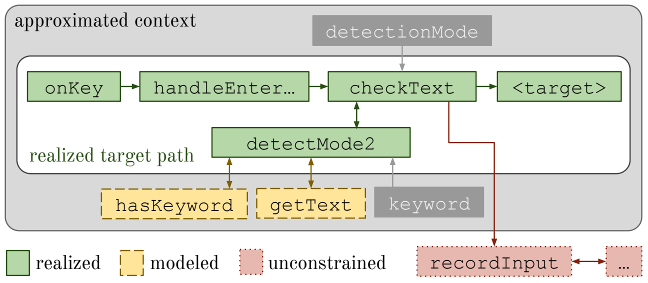

Figure 2: Separation of the target path flow based on the approximation level and scope of the constraint analysis

The approximation is determined by the scope of the static constraint analysis (i.e. the depth of auxiliary methods analyzed). Methods that are within the scope form the *realized* path and are executed in full with constrained inputs. Methods at the boundary of the scope (they are invoked by realized methods but are themselves excluded from constraint analysis) are *modeled* if there are any constraints on their return values. These constraints will be captured by the analysis performed on their realized caller methods. We treat the constrained return values as part of the program state accessed by the path and encode them in the approximated context.

All other methods that might be invoked by the target path are *unconstrained*. We execute them to ensure that any side effects are performed and can be analyzed later, such as updates to persistent storage or the scheduling of future tasks. These side effects are essentially implicit dependencies that our object-level constraint analysis does not track but that might affect the semantics of the path. One might expect that it would also be beneficial to likewise execute modelled methods and then force their return values to the constrained values. However, we find that this, similar to forced branching, results in inconsistencies between program state modified by the method and its forced return value, and can lead to excessive loss of analysis soundness. Unconstrained methods do not need to have their return values forced.

In Figure 2, which is based on a constraint analysis that extends to one level of auxiliary methods, the methods in the target's call path and the callee `v.detectMode2()` would be realized, while `v.hasKeyword()` and `v.getText()` are modeled since they must return specific values for the path's control flow. The method `v.recordInput()` is unconstrained, as the realized path imposes no constraints on its return value.

Any program state that influences the normal control flow of the realized path is handled by the context we generate from the constraint analysis. Therefore, the execution of unconstrained methods does not directly affect the realized path, except if an error occurs while the method is running. These errors, which will trigger undesired exceptional control flow, arise due to unresolved dependencies (since constraint analysis was not performed for these methods). Car handles these dependencies during execution through dynamic context refinement, which tries to recover from the errors. We describe the refinement process in Section 3.3.

The scope of the static constraint analysis and degree of path realization are configurable. Increasing it will result in greater soundness but requires more resources and can reduce coverage if the larger approximated context must satisfy constraints that cannot be resolved statically (e.g. a constrained encrypted value). Reducing the scope will result in more unconstrained methods requiring dynamic context refinement, which is based on heuristics and can lead to the execution of unsound paths. In Car, we perform constraint analysis with one level of auxiliary methods which, from our coverage and false positive results, achieves a good balance of soundness and completeness through the size of the approximated context (i.e. number of solved constraints) and the amount of dynamic recovery.

3.2 Generating an approximate context

While static constraint analysis can determine the values required to resolve a path's dependencies, concrete execution of the path requires that these values be injected into application as it is running. The dependencies can include constrained inputs and global/system state that the path accesses. To inject the constrained values for concrete execution of the target path, Car uses the context to set up the dependent state required.

Given an extracted path, Car triggers the path dynamically by directly invoking its entry-point method. The path's dependencies are resolved through a generated context that is automatically inferred from its constraints. The construction of the context is similar to the generation of input values from constraints in symbolic execution systems; however, in Car, we want to generate the context for input and non-input variables, and enforce it as the target path is executed concretely. Car uses a static phase that generates code to set up the context (which we call the path-driving framework) and a dynamic phase that invokes the generated code to produce the context at run-time. The majority of the work in constructing the initial approximated context is therefore in the static generation of the code in the path-driving framework. The generation and invocation of this code is fully automated and is performed for each target path.

The path-driving framework consists of a test harness that sets up each path with classes, methods, and fields that contain the modeled values for the path. In addition to this context, the framework will directly invoke the path's entry-point method to trigger the path.

3.2.1 Inferring and constructing the context. For each variable in a path's constraints, we infer the context by encoding a value from the solved constraints that can satisfy it—we refer to these as the enforced context value for a given constrained variable. There can be many values that satisfy the constraints and therefore many feasible contexts, though we only generate one for each path. For Figure 1, a context that can satisfy the target path's constraints might be:

$$\begin{aligned} v &= \texttt{UserInputText}\ \{\ \texttt{detectionMode} = 2, \\ &\qquad\qquad \texttt{hasKeyword()} = \mathit{true}, \\ &\qquad\qquad \texttt{getText()} = \text{"a"}\ \} \\ \mathit{code} &= \texttt{KeyEvent.KEYCODE\_ENTER} \\ \mathit{event} &= \texttt{KeyEvent}\ \{\} \\ \mathit{UserInputText} :: \mathit{keyword} &= \text{"a"} \end{aligned}$$

To enforce context values when injecting the target path, Car considers whether they are input or non-input variables and whether they are primitives or objects (we treat strings as primitive-like). The primary challenge is constraining accesses to objects. For example, to enforce the context for Figure 1, we must inject the target path with an object for the variable `v` such that when the path accesses the field `detectionMode` at Line 17 and invokes the methods `hasKeyword()` and `getText()` at Line 23, it receives the constrained context values. To accomplish this, Car generates a *constrained subclass* for each constrained object, which is a modified version of the variable's class type similar to the mock classes used in unit testing [26]. Each constrained subclass is only used within the context for one path for a particular constrained object variable. Constraints placed on members of the object are handled by controlling method return and field values. Method return values (i.e. for modeled methods) are handled by overriding the method in the constrained subclass such that they return their context values. Field members are set to their context values in the subclass's constructor when the object is initialized. An example of the constrained subclass generated for the input object for `v` in Figure 1 is shown in Figure 3(a).

3.2.2 Driving the target path. In addition to initializing constrained variables with context values (either a solved primitive value or an instantiated constrained subclass object), they must be injected with

```java
class ConstrainedUserInputText_Path0 extends
    UserInputText {
  public ConstrainedUserInputText_Path0(Context c) {
    super(c);
    this.detectionMode = 2;
  }
  @Override public bool hasKeyword() {
    return true;
  }
  @Override public String getText() {
    return "a";
} }
```

(a) Constrained subclass enforced for the input object for v

```java
public static void PathDriver0 {
  EnterKeyListener receiver = new EnterKeyListener();
  View arg1 = new ConstrainedUserInputText_Path0(null);
  int arg2 = KeyEvent.KEYCODE_ENTER;
  KeyEvent arg3 = new KeyEvent(0, 0);
  /* Constrain global heap state */
  UserInputText.keyword = "a";
  /* Inject the path by invoking its entry-point */
  receiver.onKey(arg1, arg2, arg3);
}
```

(b) Path driver method constructing the path's context at run-time

Figure 3: Path-driving framework automatically generated by Car for the target path in Figure 1

the target path. Input constraints are enforced by passing their context values as arguments to the invocation to the path's entry-point method. Non-input constrained variables include accesses to static fields and return values for static method invocations. For static fields, we explicitly set the field to its context value prior to injecting the target path. For static methods, we instrument the method to return the context value when the target path is executing and to invoke the method's original functionality otherwise.

To manage the constrained variables and context values, a central path driver method is generated for each path to construct its approximated context at run-time. When invoked, it instantiates the constrained subclasses, sets the input and non-input constrained variables, and injects the path's entry-point method with initialized inputs. For unconstrained primitive inputs, they are set to zero or an empty string. If they are unconstrained objects, they are initialized with an instantiated object of the declared input parameter type (or a concrete subtype if it is abstract), with fields set to a default zero or null value. Figure 3(b) shows the path driver method for the target path in Figure 1. During dynamic analysis, this method will be invoked in the application's process to inject the target path, which in turn will trigger the target sensitive location.

3.3 Dynamic context refinement

The approximated context set up by the path-driving framework is limited by the scope of the static constraint analysis. Unconstrained methods that are invoked by the target path, but were not analyzed, may depend on program state outside of the generated context. This can occur for the target path in Figure 1 for `v.recordInput()`, which is an unconstrained auxiliary method. Its dereference of `detectedInput` imposes a dependency since the field may not yet have been initialized. When the target path is triggered with the approximated context, execution can end prematurely in a runtime exception or crash inside `recordInput()` without reaching the target sensitive location. Car handles unresolved dependencies by refining the path's incomplete context through dynamic dependency recovery. In essence, the dependencies which could not be resolved through static means (i.e. tracked by constraint analysis and resolved in the generated context) are now handled by inferring them from the resulting dynamic dependency error. The refinement process is composed of two parts: monitoring for unresolved dependency errors and recovering from the error to return to the target path.

3.3.1 *Unresolved dependency monitoring.*

Unresolved dependency errors usually occur when an unconstrained input for the realized path (e.g. `detectedInput` in `arg1` in Figure 3(b)) is passed to an unconstrained method that does impose a constraint. The most common error is a runtime-generated `NullPointerException`, mainly due to the default null values Car uses for unconstrained variables. Other common errors are `InvalidArgumentException` and `IllegalStateException`, which are thrown by the application after a check for well-formed input. For primitive variables, a runtime `ArithmeticException` can be thrown for divide-by-zero errors. In general, technically any exception can be thrown by the application in response to unexpected (i.e. unresolved) program state. We instrument the Android runtime's (ART) exception handling code to detect when common dependency-related exceptions occur while a path driver method is executing a target path.

3.3.2 *Error recovery.*

Recovery from a dependency error requires the identification of the error's root cause, which is the dependent variable that is incorrectly unconstrained during the execution of an unconstrained method. For exceptions generated by the runtime, such as `NullPointerException`, the root cause can be identified by the runtime processes that generated the exception. For exceptions that are thrown by the application, the cause of the exception is determined by the application itself (e.g. in a conditional branch performing an error check) and the runtime environment only propagates the exception object. Therefore, to identify the root cause, information from the application's control and data flow is required. This can be provided to Car's dynamic refinement through an intraprocedural static analysis of `throw` instructions to extract their root cause variables and/or conditionals.

In the current implementation of Car, we only perform full recovery for runtime-generated exceptions where the root cause of the dependency error is available directly from the runtime. This covers the `NullPointerException`, which comprises over 75% of dependency-related errors in our evaluation. For other errors, such as `IllegalState/ArgumentException`, we perform partial recovery by suppressing the exception and continuing execution in the caller of the excepting method (i.e. oblivious to the failure [36]). As unresolved dependencies only occur in unconstrained methods, the realized path is not affected. Car's implementation can be extended to instead recover fully from these exceptions by using static information.

To recover from a `NullPointerException`, Car identifies the error's root cause from the runtime's exception processing routines, which track the register that caused the error. Car overwrites this variable with the address of an object that can resolve the dependency, which we call the *recovery object*, and transparently returns execution to the instruction where the error occurred (i.e. the *recovery location*). Since the register now contains a value expected by the application, execution continues along the target path as if the error never occurred. The recovery object does not affect the control flow of the realized target path, as dependency recovery is needed only for unconstrained methods (any variables influencing the realized path would have been captured by the constraint analysis and already resolved).

To obtain the recovery object, a seemingly obvious approach would be to simply instantiate an object based on the declared type of the root cause variable. However, it is unclear how to initialize the object properly, as constraints are not extracted for unconstrained methods. Initializing the fields of the object to null values will trigger further errors in the execution, as the application will expect objects to be initialized properly. Invoking the default constructor may set some fields to an expected default value but many application classes define a custom parameterized constructor that populate its fields from the arguments (it may even throw a further exception if these arguments are null). Instantiating other objects to populate the fields can lead to a series of instantiations due to chained object references. In the worst case, one might reinstantiate all of the objects in the application for each recovery.

Instead, Car heuristically obtains the recovery object by re-using an already instantiated object from the application. It maintains a cache of recently allocated objects and constrains the re-used object based on type compatibility with the root cause variable. By re-using objects in this way, we are essentially resolving a dependency as if the injected path had been triggered normally and preceded by its dependent paths that would have provided the dependent object. If no compatible object can be found, Car then instantiates a new object and initializes it with null values.

To complete the example from Figure 1, if `detectedInput` is null when the realized target path executes, a `NullPointerException` will be thrown in `recordInput()` at Line 29. When the exception is generated, Car will detect that a dependency error has occurred in a target path, determine that the faulting instruction is a method invocation, and that the (null) receiver object is a `List`. It will search for a previously allocated `List` object or instantiate a new concrete subclass of `List` (e.g. `ArrayList`). After generating the recovery `List` object, Car will set the register for `detectedInput` to its address and return the execution to `recordInput()`. The invocation in Line 29 will succeed and the execution will continue to the file write instruction and the target sensitive action.

4 IMPLEMENTATION

Car's implementation consists of: (1) a static context inference component to extract target paths and generate approximated contexts, (2) a dynamic driving controller to inject the paths, and (3) an instrumented version of the Android OS (AOSP) to facilitate the path driving process and perform dynamic context refinement. The static component is in Java and operates directly on the application's bytecode (APK file). It uses Soot [45] for its base static analysis and Z3 [10] for constraint solving. The dynamic controller is in Python and the instrumentation of AOSP is for Android 10.

4.1 Static targeting and context inference

We base the static extraction of target paths and constraints on the publicly available static component of Tiro [49]. We augment Tiro's constraint analysis with greater support for different types of constraints, including class type constraints and non-null constraints for instance object accesses.

For each path, Car must generate code to construct the approximated contexts and create the path driving framework. We also instrument application code at target locations to log when a target has been reached (for evaluation). We use Soot's [45] bytecode generation to construct these elements and store them with the rest of the static analysis output (we do not repackage the original APK or binary of the application).

4.1.1 Generating constrained subclasses. For each constrained object variable, the generation of its constrained subclass in the path-driving framework requires identifying the base class to extend. One might assume that this would be the constrained variable's declared type in the application bytecode, but it often cannot be directly used for two reasons: (1) the declared type is abstract and cannot be instantiated to create a context object (especially true for constraints on inputs of entry-point methods), and (2) the target path itself may assume a specific type (i.e. it imposes type constraints, like Line 8 in Figure 1). To resolve these conditions, when performing static constraint analysis, Car also extracts type constraints for variables used in class- or type-related operations, such as `cast`, `instanceof`, and any object accesses. When constructing a constrained object, it searches the application's class hierarchy tree for a concrete class that fulfills all of the type constraints on the variable. If there are multiple such classes, Car randomly chooses the most specific subclass (i.e. a leaf of the class hierarchy tree), preferring an application class over one declared by the Android framework or Java runtime library. This heuristic is based on the notion that the application has extended a class for a purpose and unconstrained methods may depend on the extra functionality.

4.1.2 Initialization of constrained subclasses. When Car generates constrained subclasses for approximated contexts, it needs to specify how they should be initialized when they are instantiated. We assume that a constrained subclass should behave in the same manner as its extended base class for unconstrained members. For each constrained subclass, we generate a constructor method that invokes the base class's constructor, which will presumably set all unconstrained fields to their initial or default values. Car heuristically chooses to invoke the base constructor method with the fewest input parameters. In cases in which the base class constructor requires input arguments, we set them to a default zero or null value if they are unconstrained by the target path. The generated constructor will set any constrained field members after the base constructor invocation to ensure the enforced context values are visible to the target path

We found that some classes are meant to be constructed in a certain way and may rely on a static initializer method to ensure all state is initialized properly. For instance, the `MotionEvent` class in the Android framework is backed by a native class that provides most of its functionality. Rather than using constructors, static initializer methods are provided to ensure that when the

Java `MotionEvent` class is instantiated, the native backend object is created as well—if the native object is not constructed, we find that segmentation faults arise later when the Java object is used. We identified several commonly used classes that are irregularly constructed and specifically invoke their initializer methods when we need to create a constrained subclass for them.

The construction of constrained subclasses can be recursive if constraints exist for chained object references. For instance, a path may require that when method `x.foo()` is invoked, it returns an object `y` where the field `y.a` contains a specific value. Car would first construct a constrained subclass for `x` that overrides method `foo()`. This method must return a constrained object for `y`, so it constructs another constrained subclass where field `a` is set to the required context value.

Our current implementation returns only one context value for each constrained variable. However, we can easily return a sequence of values for cases when the path makes multiple accesses to the same object/state (e.g. invocations to the same method) and expects a different value for each access.

4.2 Dynamic driving controller

The dynamic controller receives information from the static component, including the extracted target paths and their path driving code, and runs on a machine connected to a physical Android device. For each application, it sequentially injects each target path by sending a control message to a custom system service running within an instrumented OS on the test device. It monitors the output log to determine whether the target location for each path has been reached and to restart the application on a crash, which usually occurs when an incomplete context is inferred for a target path and our heuristics for dynamic recovery fail.

4.3 Custom Android OS

Our modifications to AOSP span ~3000 added or modified lines of code and include instrumentation of the Android framework and the Android runtime (ART).

4.3.1 Delivery of injected paths. We instrument the framework to add a custom system service to deliver injected events. Through socket and interprocedural (IPC) calls, the service invokes the path driving frameworks in the application's main thread. To load our custom path driver methods and contexts, we modify the class loader within ART. When an context class is requested, the class loader will load it from Car. We also load instrumented application classes in this way so that the code modifications occur only in memory and the application cannot detect any changes to its binary, which often trip code integrity checks.

4.3.2 Dynamic dependency recovery. Car's dynamic context refinement and recovery of dependency errors involve instrumenting ART's code interpretation processes. Monitoring requires the modification of exception handling code to detect when an exception is thrown. For recovery, we instrument locations where null exceptions are generated by ART. The recovery process is architecture dependent due to the manipulation of machine registers, including the PC. We currently implement dependency recovery for ARM and we force ART to run in ARM mode, if possible. This was not an issue with our datasets since applications containing native code usually include both ARM and ARM64 binaries for greater compatibility. Also, the generation of recovery objects requires a cache of previously allocated objects. We instrument the class initialization process to store the addresses of allocated objects in reverse chronological order.

4.3.3 Null recovery instrumentation in ART. The exact method of recovering from a dependency-related null pointer exception in ART depends on the current execution mode and whether the code is precompiled.

When ART is in the DEX "interpreter" mode, the routines for handling object access instructions contain explicit checks for a null receiver object. We add a hook into these checks to determine whether the null error is occurring during Car's path driving, which we determine by searching backward through the stack trace to find a path driver caller method. If so, the recovery process is triggered and the recovery object is returned to the original object access routine, which can now operate on the non-null receiver.

When ART is in "quick" mode, it is executing precompiled DEX code. Object accesses in this mode are handled in one of two ways: through a trampoline function to handle a virtual field or method reference (when the compiler cannot statically resolve the receiver type precisely) or a direct native memory access at the resolved field/method offset within the receiver object. For the trampoline case, explicit null receiver object checks are also present within the trampoline function and dependency recovery proceeds in the same manner as the interpreter mode.

For direct native memory accesses, a null receiver object will result in a segmentation fault when ART tries to access a field or method within. ART has a custom signal handler that will triage the segmentation fault and eventually generate a `NullPointerException` object to be thrown. We instrument the signal handler at the point when it has determined that the signal is the result of a null pointer error. We then identify the DEX instruction corresponding to the native PC where the error occurred and use the declared receiver class type for the generation of the recovery object. We determine the machine register assigned to hold the receiver variable for the DEX instruction and overwrite it with the address of the generated recovery object. Using the signal context from the original segfault signal, we restore the machine's context to the PC where the error originally occurred, with the original register values (with the exception of the register now holding the recovery object). The execution should return to the faulting instruction, with the memory access now performed on the recovery object's address rather than a null address.

5 EVALUATION

We evaluate Car on popular applications from the Google Play application marketplace to demonstrate its ability to generate contexts and trigger a wide range of target sensitive behaviors in large, complex applications. We are interested in answering the following research questions:

Q1: Does the use of context approximation and refinement improve dependency resolution for targeted execution?

Q2: Is CAR effective at reaching target code locations in Android applications?

Q3: Are the paths that CAR executes sound, despite forgoing full dependency tracking?

Q4: Can CAR uncover new security-sensitive behaviors?

5.1 Experimental setup

We ran CAR's static component on machines with Intel Xeon E5-2650 CPUs, with a JVM configuration of 200 GB of memory and 24 threads. The dynamic driving controller ran on an Intel i7-3770 machine with a tunneled USB-A connection to physical Android devices. [1] We tested on Pixel and Pixel 2 devices running our modified version of Android 10 without Google Play Services installed. We used the same device type for all of the testing for each application.

5.2 Dataset

To demonstrate CAR's generalizability on a variety of common applications, we crawled the Google Play marketplace for the binaries and metadata of all free applications in June 2019. To form our dataset, we sorted Google Play's application categories into 15 related category groups. For each group, we extracted the 25 most popular applications (determined by the total number of downloads), resulting in 375 applications in total. This set includes well-known applications such as Facebook, WhatsApp, Pokémon Go, Spotify, BBC News, Microsoft Office, eBay, Instagram, Uber, and AccuWeather.

Several applications crashed when launched on our test devices due to device incompatibilities or dependencies on Google Play Services, which is not included with AOSP. Some applications triggered errors in our version of Soot, due to incomplete support for specific DEX instruction sequences. We skipped them, resulting in an effective dataset of 310 applications.

5.3 Effectiveness of contexts for dependency resolution (*Q1*)

To evaluate CAR on a variety of sensitive behaviors and paths, we configured it to target the sensitive source and sink methods from FlowDroid [3]. While we are not performing taint analysis, the methods include a variety of different sensitive actions on Android that are likely of interest when performing security analysis. In addition, we also target invocations to reflection APIs, code loading APIs, and native methods, which may be used for obfuscation. In total, we target over 235 different APIs in our evaluation, which is likely greater than what would be targeted during realistic usage. This overapproximation of targets places greater strain on the static and dynamic analyses, as more paths must be targeted, and allows us to fully exercise CAR's abilities.

We compare CAR's ability to resolve dependencies for targeted execution against IntelliDroid [48] and TIRO [49], which are similar hybrid targeted execution frameworks for Android. TIRO is based on IntelliDroid and they employ the same target path extraction and dependency resolution techniques, which favor precise static dependency tracking. As CAR uses the initial targeting from TIRO, it also extracts the same target paths. We execute the target paths dynamically with CAR and measure the effectiveness of using contexts to resolve dependencies. We were unable to run IntelliDroid [48] or TIRO [49] as their dynamic frameworks only support Android 4.3 and 6.0, respectively, which is incompatible with our devices and our dataset (Android 4.3 also predates ART, where we implemented CAR's dynamic dependency recovery). Instead, we analyze the output from their dependency analysis and generously assume that the lack of dependency tracking errors reported by IntelliDroid/TIRO means they would have triggered the path.

Of the targets triggered by CAR, IntelliDroid/TIRO reported incomplete dependency tracking or unsupported event injection for 72.1%. These errors would have prevent the tools from executing the target paths at run-time. This would result in IntelliDroid/TIRO reaching less than a third of the targets reached by CAR—an improvement of 3.6× in target coverage.

Of the paths that IntelliDroid/TIRO could not execute, we analyzed the techniques used by CAR to resolve their dependencies. Statically generated contexts were necessary in 84.7%. Dynamic context refinement in unconstrained methods was necessary in 53.3%. This consisted of recovery from runtime-generated null exceptions, with 92.7% of the recovered paths re-using objects from the application. Other dependency-related exceptions that were only partially recovered (by suppressing the exceptional control flow) occurred in 18.1%, with `IllegalStateException`'s forming the majority of the cases (13.5%). CAR's ability to reach the targets, despite the approximations made in the static constraint analysis, shows the effectiveness of hybrid dependency resolution techniques over purely static tracking.

5.4 Triggering target locations (*Q2*)

5.4.1 Evaluation on EvaDroid. Due to lack of availability and/or incompatible Android versions, we were unable to run previous targeted or forced execution tools to compare against CAR on the Google Play dataset. Instead, we ran CAR on the EvaDroid test suite [6], which was used to evaluate several previous tools [38]. We compare CAR's results with these published numbers for DroidBot [20] (a purely dynamic exploration tool), GroddDroid [1] (a hybrid GUI exploration and forced execution tool), and IntelliDroid [48].

EvaDroid contains 22 synthetic applications representative of evasive malware. "Payloads" are hidden from dynamic analysis through complex activation conditions, such as timing conditions or device fingerprinting. We ran CAR on the EvaDroid dataset and in Table 1, we compare CAR's payload coverage with the results from the previous study. We also confirmed that CAR triggered no false positive paths on the test applications.

CAR can reach a greater number of payloads than the purely dynamic tool, DroidBot; this agrees with the results from our large-scale evaluation below. It also achieves greater payload coverage than GroddDroid, likely because GroddDroid only forces branches encountered during its initial dynamic GUI exploration, which has limited coverage. CAR also outperforms IntelliDroid, which is expected from our results in Section 5.3.

5.4.2 Large-scale evaluation. We perform a large-scale comparison of target coverage in popular applications against several state-of-the-art dynamic GUI and model-based exploration tools:

[1] Tunneling over SSH and VPN was required due to COVID-19 access restrictions for the building where our analysis infrastructure was located.

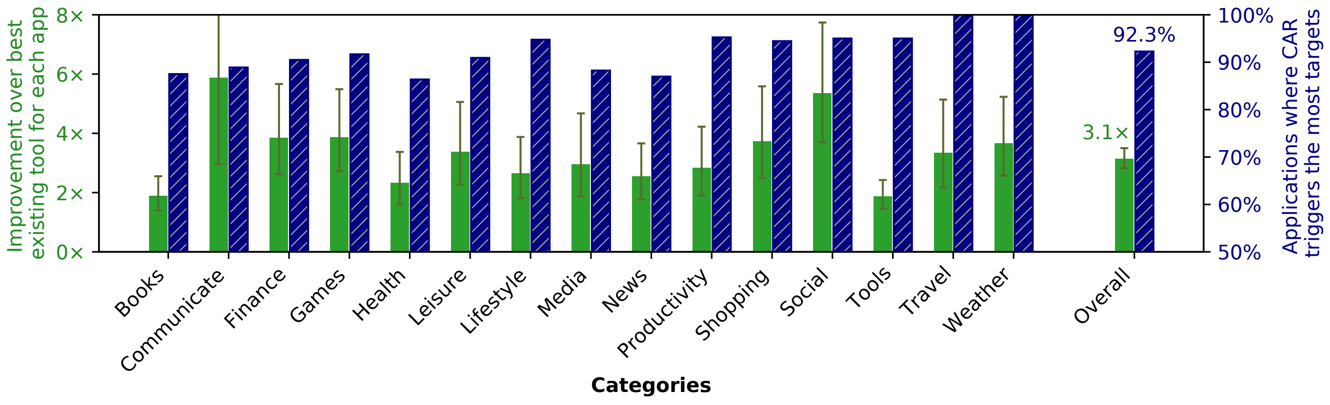

Figure 4: Comparison of non-trivial sensitive targets triggered by Car and by existing dynamic driver tools

The solid green bars (left axis) show the average improvement in the number of non-trivial targets triggered by Car against the best-performing tool (Monkey, DroidBot, or Ape) for each application. The striped blue bars (right axis) show the percentage of applications for which Car triggers the greatest number of targets of all the dynamic tools tested.

Tool		Payloads triggered
Car		73%
DroidBot	(evaluation from [38])	17%
GroddDroid	(evaluation from [38])	37%
IntelliDroid	(evaluation from [38])	33%

Table 1: Comparison of payload coverage on the EvaDroid [6] test suite of evasive applications

Monkey [27], DroidBot [20], and Ape [15]. We ran the tools for an average of three hours for each application, using the default configuration for DroidBot and Ape, and a throttle of 100 ms for Monkey similar to previous work (and gives a slight edge in its coverage [31]). To account for the delays caused by Car's dynamic dependency recovery, we conservatively used a longer wait time of 10 seconds between injections for Car (this could have been significantly reduced, as we explain in Section 5.9). We also identified the targets that were trivially triggered without any user input during the first 30 seconds when the application launched. We remove these targets in our comparisons, as they can be reached without the use of any tool.

In Figure 4, we show Car's improvement in triggering target behavior on our dataset. Car dynamically reached 125 sensitive targets for each application on average. When comparing Car to each tool individually, Car triggered 4.5× more non-trivial targets in an application than Monkey, 5.2× more than DroidBot, and 7.1× more than Ape. However, we also found that between Monkey, DroidBot, and Ape, some perform significantly better for certain applications than others. We compared Car against the best-performing tool for each application and found that we were able to reach an average of 3.1× more targets. Furthermore, Car triggered the greatest number of targets for 92.3% of the applications.

Car showed the greatest improvement in the "Communication" and "Social" categories (> 5× more non-trivial targets). These applications often require the user to log in before certain functionality can be accessed; this log-in dependency blocked the GUI exploration tools from making much progress.

In addition to the blanket coverage, we also considered which targets were triggered by Car and by the other tools. Figure 5 shows a breakdown of targets triggered in an application by the various tools, including those trivially triggered during the application's launch. A large portion (44.2%) were reached only by Car.

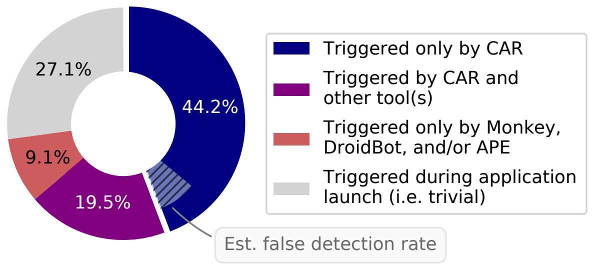

Figure 5: Average breakdown of the sensitive targets triggered by the different dynamic tools

5.5 False positives (*Q3*)

Of the 44.2% of newly triggered targets in Figure 5, we manually inspected a subset to determine whether they are truly reachable or whether they were the result of unsound path execution (i.e. false positives). Our sample set was chosen as follows: (1) for each category, we randomly sampled five applications; (2) for each application, we computed the list of targets that were triggered by Car but not by any other tool; (3) we randomly chose one of those targets and analyzed the decompiled code that would trigger it. In total, we manually inspected 75 target paths.

Our primary objective was to determine whether a target newly triggered by Car would be executed during normal execution. From our experience, applications often include third-party libraries in their APKs but may not use all of the code within them. Since Car invokes path entry-point methods directly, it is possible that some of the injected events would not have been registered or triggered during normal execution. For each target in our sample set, we checked whether all of the invocations, control flows, and data flows were valid across the path methods. We also checked whether the path's entry-point is live. For entry-point event handlers that must be registered with the framework, we recursively checked the path(s) to the registration call-sites.

We determined that 18.7% of the sampled targets were reached through infeasible paths, where the samples were drawn from the subset of targets that were triggered only by Car. When we translate this rate across all of the targets triggered by Car and assume that overlapping targets that were also reached by Monkey, DroidBot, and/or Ape are true positives, we have a false detection rate of 9.0%. This was heavily skewed by the targets sampled from the "Games" category, where 4 out of 5 inspected targets were determined to be dead code. For the rest of the categories, the average number of false positives found was 0.7 out of 5.

We found the underlying reason to be static imprecision in the points-to analysis when extracting target paths. This manifested in two ways: (1) the entry-point analysis incorrectly identified event handlers as application entry-points due to the conservative aliasing of objects at registration call-sites; and (2) method invocation edges were incorrectly constructed in the call-graph due to conservative aliasing of the invocations' receiver objects. All the false positives found were located in first- or third-party libraries packaged with the applications as determined by the package name of methods in the paths. The spike of false positives within the "Games" category is likely due to the large number of libraries used, including graphics rendering, analytics, and advertisement libraries. Libraries introduce a great deal of code that must be analyzed but may not necessarily be used by the main application, increasing the effects of conservatism due to imprecision.

5.6 Analysis of newly triggered targets

During our manual analysis, we also enumerated the different reasons why the other dynamic tools were unable to trigger the cases where the target was a true positive for Car (i.e. not a false positive from the previous section). For 21%, the target could not be reached by the other tools because the application was blocked by a login screen. For a further 15% of cases, the application was blocked by a dialog for Google Play Services. For 13%, the target could only have been reached for certain devices or versions or if a specific error, such as a network failure, had occurred. While these conditions could not have been achieved in our test environment, Car's inferred path contexts was able to mimic the environment required.

For the remaining 51%, we found that they could have been reached by normal UI interactions or system event injections. However, a few would have required extremely complex interactions between event paths. In one case, the target occurs after the user purchases a travel package through the application, is located at an airport, and has Uber installed on their device. The conditions required for this path include UI flow across multiple screens (purchase process), input that is difficult to generate (purchase information), specific system state (location), and dependence on other installed applications. Car directly invoked the location event handler for the target path, thus bypassing the multiple purchase-related requirements, and used the path's context to return `true` when the application queried the framework for the other application.

Sensitive functionality	Google Play			Creepware		
	Apps	Calls	Cxt.	Apps	Calls	Cxt.
Location	84	237	89%	14	42	86%
Personal data	6	6	83%	4	4	100%
Media	3	3	100%	1	1	100%
Telephony	7	11	100%	4	4	50%
Network	59	99	74%	11	20	85%
Files	220	1199	88%	63	361	10%
Databases	76	187	90%	11	47	94%
Package mgr.	134	222	93%	12	21	90%
Reflection	169	447	84%	25	113	75%
Code loading	7	7	71%	10	10	60%
Native code	223	1090	88%	29	120	68%

Table 2: New sensitive behaviors missed by other tools that only Car could find (and the percentage requiring contexts)

5.7 Analysis of sensitive behaviors (*Q4*)

The new behaviors uncovered by Car ranged over a variety of different security-sensitive actions missed by the other tools. In Table 2, we break down the sensitive actions that only Car was able to find. We further show that Car's context-based techniques were essential and measured the percentage that required contexts or dynamic dependency recovery. Since our original dataset contained only benign applications, we also compared Car and Monkey on a set of 91 malicious applications recently labeled as evasion or surveillance "creepware" [37]; these were used for interpersonal attacks, such as harassment or stalking.

Car operates effectively on both benign and malicious applications and uncovers new sensitive or malicious behaviors missed by the other tools. This ranged from recurring location accesses that leak location data to the network, to accesses to local email accounts hidden under misleading UI. We also found cases in which code was dynamically loaded on paths that only Car was able to trigger.

To verify that Car is able to uncover sensitive behaviors in malicious applications, we ran Car on the Creepware [37] dataset and find it can detect a variety of malicious behaviors that were manually confirmed to be true positives. We provide an analysis of these cases below to demonstrate Car's ability to uncover malicious or evasive activity in applications, by guiding or targeting execution toward sensitive actions. While a dynamic taint tracking tool would have been ideal for detecting the cases of private data leakage, we were unable to integrate Car with previous taint tracking tools because they were implemented for older versions of Android or their implementation was not fully available [11, 42, 54] We instead relied on manual analysis and instrumentation to determine the flow of data triggered by Car's targeting.

Periodic background location tracking: One malicious application was intended to surreptitiously track the location of an unsuspecting victim. It uses Android's alarm service was used to periodically invoke a location gathering method, which regularly accesses the device's location data and send it to the network. Car was able to trigger the periodic location gathering function and execute this behavior by directly invoking the alarm callback method and using contexts to resolve the dependencies and constraints imposed by the callback path. In contrast, Monkey was unable to access this functionality due to log in and set up requirements that were necessary to enable the tracking. The leakage of location data in the background is performed without user awareness and should be triggered for dynamic security analysis to evaluate whether an application violates the privacy of the device's user.

Location tracking on a location change: There are several methods of accessing location information, including a callback-based approach. One application registers a location callback that is invoked by the framework when the device's location has changed. The application then stores the new location information into a cloud storage location and sends it to the network asynchronously. Car was able to execute the location leakage path while Monkey was unable to trigger the location change required for the callback to be activated.

Leakage of private data to the network: A large number of the tested malicious applications (approximately half) are surveillance applications that registers the device for a tracking service. This tracking enables the device's location, phone number, identifiers, photos, videos and/or received messages are sent to a third-party (supposedly for emergency situations, though this functionality can also be used for spying). In some of the applications, Monkey was able to trigger the private data leakage. However, in at least two of the tracking applications, Car was able to trigger private data leakage of device identifiers and location data to the network that Monkey was unable to reach. This was primarily due to a requirement to register the device with the tracking service, which Car was able to bypass.

Transmitting microphone recording over Bluetooth: A spying application streams input from the microphone to a Bluetooth headset, allowing someone to eavesdrop on the device's surroundings and on the user's conversations. The functionality is only accessibly if a Bluetooth headset is connected to the device. Other dynamic tools would have difficulty triggering the spying behavior unless the analysis environment specifically includes such a headset. Car used its contexts to resolve the system Bluetooth dependency.

Hidden access to device accounts: One application provided a "cloning" functionality, in which the application's main activity shows a set of public accounts and a secondary hidden activity enables access to a set of private accounts. We manually confirmed that Car was able to access both interfaces and detect access to both sets of accounts. The hidden interface would have been particularly difficult for dynamic tools such as Monkey, as there were complex UI actions required to make the activity visible. In contrast, Car was able to trigger the intentionally hidden account functionality. Similarly, we also found similar evasive applications that hide an account-related activity behind a calculator interface. The hidden accounts are accessible only when a specific password is entered into the calculator. Car was able to explore past the initial calculator interface without having to decipher the correct password.

5.8 False negatives

While there is overlap in the targets reached by Car and the other tools, we did find that 9.1% of all triggered targets in Figure 5 could be reached by at least one of the other tools but were missed by Car. We consider these known false negatives of Car. We manually analyzed 15 of these targets across all of the categories and found the following underlying reasons: (1) incomplete object handling across Android interprocedural (IPC) invocations; (2) incomplete handling of constraints, especially for lists and arrays; (3) reliance on the stack trace for dependency error monitoring, which misses errors in new threads as they have a new stack; (4) constraints on the results of one-way computations, such as encryption or hashing, which could not be inversed to generate a context value; and (5) any-path extraction of target paths, which may extract infeasible paths to a target due to the conservative call-graph and miss a true path.

Items (1) – (3), which applied to over 75% of the sampled false negatives, are artifacts of the implementation and can be fixed with more engineering. They were the primary reason why Car produced lower coverage in a few applications. (4) is a fundamental limitation of symbolic analysis, although in some cases, it was mitigated when the required result of a one-way computation can be statically extracted and set by a modeled method. (5) can be mitigated by sampling more paths for each target, though the extraction of all paths to a target would ultimately be exponential.

5.9 Performance

We measure the performance of the static and dynamic components of Car separately on our Google Play dataset. We ran the static context inference analysis with a timeout of 240 minutes and retrieved partial results in cases in which full analysis had not completed. For most applications, the full time was used. For the dynamic component, we injected all of the paths extracted by the static analysis (maximum of 2700 for each application). We throttled Car execution with a conservative wait time of 10 seconds for a target to be reached before injecting the next path. On further analysis, this could have been significantly reduced as an average path took 1.4 seconds to reach the target location.

We examined the sensitive targets triggered over the duration of the analysis for each tool and plotted the cumulative number of triggered targets in Figure 6. While only a portion of the statically extracted targets were dynamically triggered by the tested tools, we suspect that many of the unreached targets were actually infeasible due to static imprecision. Car steadily made progress as it injected each statically extracted path. For the others, while they were successful at finding targets near the beginning, they made significantly less progress over time. After an hour of analysis, Car already shows a large improvement in the number of triggered targets.

6 DISCUSSION AND LIMITATIONS

As stated in Section 5.8, a fundamental limitation is that some constraints cannot feasibly be solved. Furthermore, injecting path entry-point methods directly can lead to the execution of paths

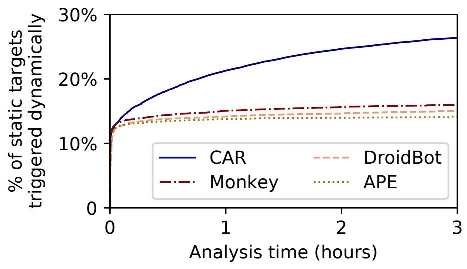

Figure 6: Cumulative number of targets triggered over time

that might not occur normally if their entry-points were never registered with the framework. Because the dynamic dependency recovery refines the context heuristically, it may also construct infeasible contexts that would never occur during normal execution (by returning an incorrect recovery object). All of these can affect the soundness of the paths triggered; however, these issues are not unique to Car. Forced execution tools will also execute infeasible paths from unreachable entry-point methods and incorrect data accesses, especially as they enforce branch outcomes without ensuring that the accompanying data dependencies are resolved. Furthermore, without analysis of how data values are used, forced execution can result in infeasible control-flow paths due to the forcing of a normally impossible combination of branch outcomes.

Car's constraint analysis ensures that the extracted path is consistent with respect to the data it accesses; the context will either contain values that satisfy all of the constraints imposed by the path's conditional branches or, if the branches required are incompatible and the path is infeasible, no context would be generated. We believe Car achieves a good balance between its sources of unsoundness and its coverage of target locations, demonstrated by its ability to outperform existing dynamic tools in target coverage while maintaining a 9.0% false detection rate on popular applications. In addition, our false positive evaluation in Section 5.5 showed that the primary source of infeasible paths in Car were due to static imprecision (i.e. over-conservative entry-point detection and points-to analysis), which also affect forced execution.

7 RELATED WORK

Car is most closely related to other targeted dynamic analysis tools. IntelliDroid [48] and Tiro [49] perform targeted execution of Android applications, though they rely on precise static dependency tracking and resolution. Similarly, guided symbolic execution tools such as AppIntent [53], WatSym [30], and [5] implicitly handle dependencies by modeling the program state symbolically. However, the resources required to precisely track and resolve the program state either statically or dynamically can ultimately reduce the coverage of targets due to overhead. Furthermore, guided symbolic execution cannot easily integrate with dynamic analysis tools that require concrete execution of the application. GroddDroid [1], Harvester [34], DirectDroid [46], and Ares propose forced execution, which bypasses a path's constraints on its dependencies by enforcing specific branch outcomes. Forced branching can lead to unsound or infeasible paths, resulting in false positives.

Car's static constraint analysis is related to symbolic execution (such as EXE [9] and KLEE [8]) and concolic execution (such as DART [13], CUTE [39], and ACTEve [2]). Rather than tracking constraints dynamically, which adds significant overhead, Car uses hybrid dependency resolution. UC-KLEE [32], which performs under-constrained symbolic execution by invoking arbitrary methods directly, is in a way bypassing the dependencies of the path. The preconditions imposed by a method on its inputs [18, 32] are similar to contexts. We can achieve more sound execution by executing paths rather than individual methods. Car is also similar to chopped symbolic execution [44], in which irrelevant function calls are skipped. Chopped functions are similar to Car's modeled methods and any unconstrained methods that are aborted.

Car is a hybrid tool and is related to other static-guided dynamic analysis tools. Brahmastra [7], AppDoctor [17], SmvHunter [40] and SmartDroid [57] guide execution by driving component and UI transitions, which are more coarse-grained than code paths, and they do not handle dependencies between event paths. ContentScope [58] generates inputs for paths in content providers and also does not handle inter-path dependencies. AppAudit [50] and AppIntent [53] use static analysis to guide approximate or symbolic execution, respectively, for the verification of static information flows. In contrast, Car guides concrete execution and can be integrated with general dynamic analyses. Car's static component is also similar to other static Android tools, such as FlowDroid [3], Epicc [29], Apposcopy [12], Amandroid [47], CHEX [21], and [52].

Car's aims to drive the execution of Android applications, similar to testing frameworks such as DynoDroid [22], EvoDroid [23] and Sapienz [24], and model-based explorers such as Ape [15], Stoat [41], DroidBot [20], TrimDroid [25], and A^{3}E [4]. Fuzzing tools, such as AFL [55], are also commonly used outside of Android. They aim for full coverage, while Car focuses on coverage of target locations. The techniques we use to focus execution to target paths enables greater coverage of locations of interest to an analyzer. Semi-targeted exploration, such as FuzzDroid [35] and Xdroid [33], inject system/framework values, which also helps resolve dependencies on framework state. CrashScope [28] dynamically triggers application crashes by exploring different system configurations; however, it does not handle dependencies on data within the application. Similarly, generating useful test inputs, such as TextExerciser [16], is also a form of guided fuzzing.

8 CONCLUSION

We present Car, a hybrid approach to dependency resolution through context approximation and refinement. We use a combination of static constraint analysis and dynamic error recovery to resolve dependencies on program state while mitigating the trade-off that must be made between precision, completeness, and scalability in static dependency tracking. We applied Car to the targeted execution of Android applications and were able to reach 3.1× more non-trivial sensitive targets for an application with a false detection rate of 9.0%. These sensitive behaviors, which would have otherwise been missed, are essential for the security analysis of Android applications.

ACKNOWLEDGMENTS

We would like to thank He (Shawn) Shuang, Mingyue (Shirley) Yang, and Ivan Pustogarov for their suggestions and feedback on this work. We also thank Kevin Roundy for sharing their Creepware dataset. We thank the anonymous reviewers for their constructive comments. The research in this paper was supported by an NSERC CGS-D scholarship, NSERC Discovery Grant RGPIN-2018-05931, and a Tier 1 Canada Research Chair.

REFERENCES

[1] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, J-F Lalande, and V Viet Triem Tong. 2015. GroddDroid: A gorilla for triggering malicious behaviors. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE, 2015)*. IEEE, 119–127.

[2] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. ACM, 59.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Drew McDaniel. 2014. 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)*. ACM, 259–269.

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013)*. ACM, 641–660.

[5] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, 12–22.

[6] Luciano Bello and Marco Pistoia. 2018. ARES: Triggering payload of evasive Android malware. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft@ICSE 2018)*. ACM, 2–12.

[7] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 2014)*. USENIX Association, 1021–1036.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*. USENIX Association, 209–224.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2007. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2007)*. ACM.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*. USENIX Association, 393–407.

[12] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 576–587.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM, 213–223.

[14] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*. ACM, 281–294.

[15] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE 2019)*. IEEE/ACM, 269–280.

[16] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. 2020. TextExerciser: Feedback-driven Text Input Exercising for Android Applications. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*. IEEE, 1071–1087.

[17] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys 2014)*. ACM, 18:1–18:15.

[18] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*. Springer, 553–568.

[19] Patrik Lantz and Anthony Desnos. 2011. DroidBox: An Android application sandbox for dynamic analysis. https://www.honeynet.org/projects/active/droidbox/. Accessed: June 2020.

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion Volume (ICSE-C 2017)*. IEEE, 23–26.

[21] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS 2012)*. ACM, 229–240.

[22] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 21st ACM SIGSOFT International Symposium on the Foundations of Software Engineeringg (FSE 2013)*. ACM, 224–234.

[23] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 599–609.

[24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 94–105.

[25] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE 2016)*. IEEE/ACM, 559–570.

[26] Mockito 2020. Mockito. https://site.mockito.org/. Accessed: June 2020.

[27] Monkey 2020. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. Accessed: June 2020.

[28] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*. IEEE, 33–44.

[29] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013)*. USENIX Association, 543–558.

[30] Riyad Parvez, Paul AS Ward, and Vijay Ganesh. 2016. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON 2016)*. IBM/ACM, 116–127.

[31] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how I learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test (AST@ICSE 2018)*. ACM, 34–37.

[32] David A Ramos and Dawson R Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 2015)*. USENIX Association, 49–64.

[33] Bahman Rashidi and Carol Fung. 2016. Xdroid: An Android permission control using hidden Markov chain and online learning. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS 2016)*. IEEE, 46–54.

[34] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society.

[35] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory behave maliciously: Targeted fuzzing of Android execution environments. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE 2017)*. IEEE/ACM, 300–311.

[36] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI 2004)*. USENIX Association, 303–316.

[37] Kevin A Roundy, Paula Barmaimon Mendelberg, Nicola Dell, Damon McCoy, Daniel Nissani, Thomas Ristenpart, and Acar Tamersoy. 2020. The Many Kinds of Creepware Used for Interpersonal Attacks. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*. IEEE.

[38] Aleieldin Salem, Michael Hesse, Jona Neumeier, and Alexander Pretschner. 2019. Towards Empirically Assessing Behavior Stimulation Approaches for Android Malware. In *Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2019)*. IARIA XPS Press, 47–52.

[39] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[40] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014*. The Internet Society.

[41] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 245–256.

[42] Mingshen Sun, Tao Wei, and John Lui. 2016. TaintART: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 331–342.

[43] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society.

[44] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 350–360.

[45] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*. IBM, 13.

[46] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. 2018. Automated hybrid analysis of Android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing* 18, 12 (2018), 2768–2782.

[47] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*. ACM, 1329–1341.

[48] Michelle Y Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society.

[49] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, 1247–1262.

[50] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective real-time Android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*. IEEE, 899–914.

[51] Lok Kwong Yan and Heng Yin. 2012. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*. USENIX Association, 569–584.

[52] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*. IEEE/ACM, 89–99.

[53] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and Xiaoyang Sean Wang. 2013. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*. ACM, 1043–1054.

[54] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. Taint-Man: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices. *IEEE Transactions on Dependable and Secure Computing (DSC)* 17, 1 (2017), 209–222.

[55] Michal Zalewski. 2020. AFL. https://lcamtuf.coredump.cx/afl/. Accessed: June 2020.

[56] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*. ACM, 611–622.

[57] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: an automatic system for revealing ui-based trigger conditions in Android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2012)*. ACM, 93–104.

[58] Yajin Zhou and Xuxian Jiang. 2013. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society.