In Search Of Shotgun Parsers In Android Applications

Katherine Underwood
University of Calgary
Calgary, Canada
Email: kaunder@ucalgary.ca

Michael E. Locasto
SRI International
Infrastructure Security Group
Email: michael.locasto@sri.com

I. Abstract

In any software system, unprincipled handling of input data presents significant security risks. This is particularly true in the case of mobile platforms, where the prevalence of applications developed by amateur developers in combination with devices that hold a wealth of users' personal information can lead to significant security and privacy concerns. Of particular concern is the so-called shotgun parser pattern, in which input recognition is intermixed with input processing throughout the code base. In this work, we take the first steps toward building a tool for identification of shotgun parsers in Android applications. By extending the FlowDroid framework for static taint analysis, we are able to quantify the spread of untrusted data through 55 applications selected from 15 categories on the Google Play store. Our analysis reveals that on average, most untrusted input propagates a relatively short distance within the application code. However, we also find several specific instances of very long data propagations. In addition to providing a first look at the "state of affairs" in a variety of Android applications, our work in this paper lays the groundwork for more precise shotgun parser signature recognition.

II. Introduction

The Android platform has the largest market share of mobile device operating systems worldwide [1]. However, the nature of the Android development ecosystem means that many popular applications are created by amateur developers, who may not always follow security best practices. Additionally, the widespread use of third party libraries means that a single bug in a single library can have security implications in a huge number of popular applications, as was seen with the recent "Stagefright" vulnerability[2].

In particular, significant security vulnerabilities can be introduced if input data is not handled in a principled manner. This project addresses the problem of unprincipled input handling from a language-theoretic security, or LangSec, approach. LangSec is a relatively new sub-field of security research focused on treating program inputs as formal languages to be formally recognized. In the overview by Bratus et al. [3], four LangSec "anti-patterns" are identified, which exemplify the poor practices that LangSec seeks to prevent. These include ad-hoc notions of input validity, parser differentials, mixing of input recognition and processing, and ungoverned development/language specification drift. For reasons of computational tractability, this work focuses on identifying instances of mixed input recognition and processing—so-called "shotgun parsers"—in Android application code. Informally, the term shotgun parser refers to code that mixes input data recognition and input data processing [3] (this pattern is so named because raw input data ends up scattered throughout the code base, as if shot like pellets out of a shotgun). We present three hallmarks of the shotgun parser pattern: input spread relative to application size, number of variables involved in each tainted path, and use before full recognition. In the following sections we formalize these definitions, and carry out the first steps to quantify the prevalence of this pattern in popular Android apps.

Our analysis makes use of FlowDroid [4], which is an existing open-source library for static taint analysis of Android applications. By using FlowDroid to extract information about tainted paths from application control flow graphs, we quantify "how far" input data propagates within an application. This data serves as a jumping-off point for assessment of shotgun parser prevalence in any Android application. We also draw conclusions about input data handling patterns that exist for different types of input sources (e.g. input from screen vs. input from network card).

III. Contributions

- Path length data collected thus far can be used to classify "worst offenders" in terms of individual apps, app categories and source types. These data points can then be used as a starting point for future analysis.
- Our modifications to the FlowDroid framework can be used as a foundation for future, more in-depth analysis focused on capturing specific operations executed along long tainted paths.
- This work contributes to the relatively new LangSec approach to security analysis, and aims to contribute to improving security best practices for Android developers by approaching security analysis from a LangSec-based angle.

IV. Background and Related Work

Since Google released the first version of Android in 2007, there has been significant research into the security properties of both individual Android apps and the Android operating system itself. This work ranges in scope from surveys of Android user attitudes and behaviours, to examination of reviews posted on the Google Play store, to static and dynamic explorations of the code base. Some of this prior work is briefly discussed here.

A. Various Tools For Security Analysis

The continuing interest in Android security research has given rise to the development of a number of tools for assessing various properties of Android applications. Tools such as Stowaway [5] and PSCOUT [6] perform static analysis to empirically determine the mapping between API calls and required permissions. Android applications run on a Dalvik virtual machine, and as such the low-level code found in an Android `.apk` file is Dalvik byte code. Dexpler [7] is a tool for converting low-level Dalvik bytecode into Jimple, which is an intermediate-level representation for Java. Various tools exist for analyzing the execution of Android apps, both statically and dynamically. These include SymDroid, a symbolic execution framework for Dalvik bytecode [8]; DynoDroid, which generates app inputs for event-driven testing [9]; VetDroid, which uses dynamic analysis to capture permission use behaviour [10]; and TaintDroid, which performs real-time taint tracking as an app is running [11]. The 2013 work by Egele et al. in *An Empirical Study of Cryptographic Misuse in Android Applications* statically analyzes 11,748 apps and concludes that 88% of these use cryptographic APIs incorrectly, that is, in a manner which violates IND-CPA[1] security. This analysis is conducted using the author's Cryptolint tool, which operates on Dalvik bytecode by computing the super control flow graph of each app, and then using static program slicing to trace backwards from each crypto API call.

Of significant relevance to this project is the Soot framework [12], which was developed by the Sable Research Group at McGill University for the analysis and instrumentation of Java code. More specifically, this project relies on an extension to Soot called FlowDroid [4], [13], [14], which facilitates analysis of Android applications specifically. The operation of FlowDroid is discussed in detail in the Methodology section. FlowDroid makes use of the output of SuSi, a tool for automatically classifying sources and sinks present in the Android operating system [15].

B. Formal Language and Control Flow Graph-Based Work

The focus of this project lies in the less-studied area of formal language-based security. Some work has certainly been conducted in similar directions, however none from a specifically LangSec based angle. In *Language-based Security on Android*, the author presents a formal language to describe Android applications abstractly [16]. This work differs from ours in that we will focus on how application *input* is propagated, rather than formal specification of the apps themselves. The KTH Royal Institute of Technology in Stockholm, Sweden, have developed a tool called TreeDroid [17], which makes use of the TaintDroid framework and uses a lambda calculus-based approach to characterize and enforce safe data processing policies. This work is similar to our project in that it seeks to quantify "good behaviour" for input processing. However, the focus of our project is on statically assessing the prevalence and severity of one specific anti-pattern (the shotgun parser) in the existing set of popular applications. By contrast, the TreeDroid paper focuses on real-time monitoring and policy enforcement. An interesting extension to our work could involve enforcing a "safer" input processing policy in cases where a shotgun parser has been identified. Such work would also be a natural extension of the TreeDroid approach and in this case the KTH group's work in this area would be highly relevant.

Our work is also heavily based on analysis and traversal of program control-flow graphs. In *Generalized Vulnerability Extrapolation Using Abstract Syntax Trees* [18], Yamaguchi et al. represent application code as abstract syntax trees, and identify vulnerabilities based on structural patterns within these trees. This work is extended in the 2014 paper *Modelling and Discovering Vulnerabilities with Code Property Graphs* [19], which incorporates analysis of control flow graphs and program dependence graphs. While there are some general similarities to our work, the Yamaguchi papers focus on vulnerabilities in the Linux kernel [19], and in open source projects including LibTIFF, FFmpeg, Pidgin and Asterisk [18], whereas we focus specifically on Android applications. In addition, their approach involves developing templates for common vulnerabilities which can be compared against a code base. By contrast, our approach is to statically analyze the control flow graph of each app in our test set to gather empirical data about shotgun parser severity and prevalence.

C. The Shotgun Parser Anti-Pattern

In *Shotgun parsers in the cross-hairs* [20] and *From 'Shotgun Parsers' to Better Software Stacks* [21], Patterson et.al describe the "shotgun parser" design anti-pattern, characterized by use of input data before full recognition. In these talks, the presenters highlight various examples of how a failure to fully recognize input data leads to exploitable vulnerabilities. In the sections that follow, we expand upon this concept and propose a graph-based framework for identification of the shotgun parser anti-pattern.

V. Methodology

A. Static Taint Analysis

We frame our approach in the context of static taint analysis. Informally, taint analysis involves defining data from an untrusted source to be "tainted". When tainted state is involved in a program statement, the taint may be propagated on to other variables in that statement. For example, in a case where a variable x is tainted, execution of the statement $y = x + a$ would

[1] Indistinguishability under chosen plaintext attack

result in y becoming tainted as well. Formally, at each program statement s the set of incoming tainted state $\mathcal{T}$ is transformed to a set of outgoing tainted state $\mathcal{T}'$ according to a predefined set of taint propagation rules or transfer functions.

To describe the taint transfer functions used in our analysis, we follow the description in [13] and references therein. The transfer functions used in our analysis are as follows:

- Normal flow function - applies to straight line control flow from one statement to another. Since new taints can only be generated as the result of an API call to a source method, for normal flows $t \in \mathcal{T}' \implies t \in \mathcal{T}$. The converse is not necessarily true however, since the statement over which the normal flow function is applied may not pass on any taint from the incoming set $\mathcal{T}$.
- Call flow function - applies to statements in which one method is called from another. In this case, any tainted state from the caller method which is passed as an argument is defined to be tainted in the callee method as well.
- Return flow function - applies to statements in which one method is returned to from another (following a call). Similar to the call flow function, here any tainted state in the callee method which is returned to the caller method is defined to be tainted in the caller method upon return.
- Call-to-return flow function - captures any flows between a call and return statement within the same method. This is necessary to preserve state which was tainted before a method call, but which may not be passed as part of the method call.

In general, the set of outgoing taints $\mathcal{T}'$ is given by $\mathcal{T}' = \mathcal{T} \cup \mathcal{T}_{new} \setminus \mathcal{T}_{killed}$, where $\mathcal{T}_{new}$ is the set of all new taints generated as a result of the statement, and $\mathcal{T}_{killed}$ is the set of all taints killed as a result of the statement (i.e. taints which do not propagate).

The FlowDroid developers also define some more subtle taint propagation rules for handling of special cases such as native calls, however a discussion of this logic is not required in order to describe the basis of our analysis[13].

B. Defining the Shotgun Parser

Since the purpose of this work is to identify the prevalence of shotgun parsers in Android applications, it is necessary to define what constitutes a shotgun parser. Informally, code that contains a shotgun parser mixes input recognition and processing [3], and does so throughout the code base. Conversely, code that definitely does not constitute a shotgun parser would immediately validate all input data and, upon validation, proceed by populating strongly typed data structures with the necessary input bytes. With these considerations in mind, we propose three characteristics of a shotgun parser. Each of these characteristics is a necessary (but not sufficient) condition for the presence of a shotgun parser.

1) Spread Relative to Size
   Once data is read into an application from an external source, how far does this input data propagate though the code? Consider a control flow graph G for an application which reads in some piece of input data x. As per the principles of taint analysis, all vertices of the control flow graph in which x is involved are now considered to be tainted. We define the tainted subgraph P_x as the connected subgraph induced by the vertices tainted by x. We then quantify the spread relative to size by comparing the diameter of P_x to the diameter of G. If the diameter of P_x is comparable to that of G, this is a strong indicator for the presence of a shotgun parser. The greater the number of relatively large tainted subgraphs present in control flow graph G, the stronger the evidence for the presence of multiple shotgun parsers.
2) Large Relative Number of Variables Involved In Each Tainted Path
   Multiple distinct variables involved in a single tainted path is another indicator for a shotgun parser. In particular, if the number of variables involved in a given tainted path is large compared to the total number of distinct variables in the code, this is a significant indicator for the presence of a shotgun parser.
3) Use Before Full Recognition
   As discussed above, in the best-practice scenario, input data is fully recognized and validated before being used to populate native data structures for further operations. That is, the "recognition and validation" phase should be described by a well-defined formal grammar, and the "use" phase should involve well-defined access to a strongly typed data structure. In the worst-practice scenario, the validation of input data x is intermixed with the use of x (or, no validation occurs at all). Such ad hoc input processing is evidence that the development process of this code was not governed by a strict notion of a formal input language, and constitutes further evidence for the presence of a shotgun parser.

To summarize, the "worst case" shotgun parser would be code which exhibits all three of these characteristics in abundance. That is, many execution paths are tainted, each tainted path involves a relatively large number of variables, the relative length of each tainted path is long, and tainted variables are read from and written to in an arbitrary order.

C. Path Length Analysis

The goal of this project is to lay the foundations for development of a tool which can identify shotgun parsers in arbitrary Android applications. This is an extensive task, and so in this initial stage we focus on identification of the first characteristic only (spread relative to size). We quantify spread relative to size by using a static taint analysis approach to compute the length of the tainted path corresponding to each input source. This is described in detail in the following sections.

D. FlowDroid

1) FlowDroid Overview: Our analysis extends FlowDroid [4], a third-party open-source tool for analyzing Android applications, developed by the Secure Software Engineering group at the European Center for Security and Privacy by Design. FlowDroid itself is an extension to the Soot framework [12] for static analysis of general Java applications, specifically tailored to the analysis of Android applications.

The nature of the Android lifecycle methods means that the various operating system callbacks could be invoked at any time and in any order, as illustrated in Figure 1. Thus there is no single clearly defined code entry point, as there would be in a "traditional" piece of software.

To enable static analysis in this environment, FlowDroid operates by first constructing a dummy main method, which models the Android application life cycle by invoking all possible life cycle methods in all possible orders [4]. Next, FlowDroid generates an inter-procedural control flow graph, and uses this to track taints from a set of pre-defined sources (e.g. a network device API) to a set of pre-defined sinks (e.g. a memory location). All analysis is static. Identified flows between sources and sinks are then returned as output.

Each time FlowDroid encounters a taint (i.e. data which originates from a defined source) to be propagated, information about the tainted path is stored in an object along with relevant metadata as a flow fact object. Taint propagation entails applying one of the four predefined propagation rules described in the *Static Taint Analysis* section to the current flow fact and current statement. The output of each propagation is a set of zero or more additional flow facts. Consider a flow fact F which is propagated over a statement s. Then for the output set of flow facts $\mathcal{F}_{out}$, we have four cases:

- s does not result in any taint propagation: $\mathcal{F}_{out} = \emptyset$
- s propagates the original taint but does not create any new taint: $\mathcal{F}_{out} = F$
- s generates one or more new taints but does not propagate the original taint: $\mathcal{F}_{out} \neq \emptyset$, $F \notin \mathcal{F}_{out}$
- s propagates the original taint and generates one or more new taints: $\mathcal{F}_{out} \neq \emptyset$, $F \in \mathcal{F}_{out}$

2) Source Configuration: FlowDroid relies on an input list of predefined source and sink methods in order to perform taint tracking. Following the approach of the authors in [4], we used the output of the SuSi tool [15] to define our sources/sinks for input to FlowDroid. SuSi is a tool which uses a machine learning approach to identify sources and sinks in the Android API. We thus follow reference [15] and define a *Source* as any call into a resource method which returns non-constant values into the application code. A *resource method* is defined as any method which reads data from or writes data to a shared resource [15]. Note that these definitions mean that we treat data as tainted if it originates from anywhere outside of the application code, even if the origin point is elsewhere on the same physical device and not from an external origin point such as a network socket. This is in fact desirable, as malicious activity on a mobile device may originate from a compromised system component or other application, as in a permission re-delegation style attack[23]. Therefore data should not be treated as "safe" simply because it originates from elsewhere on the same physical device.

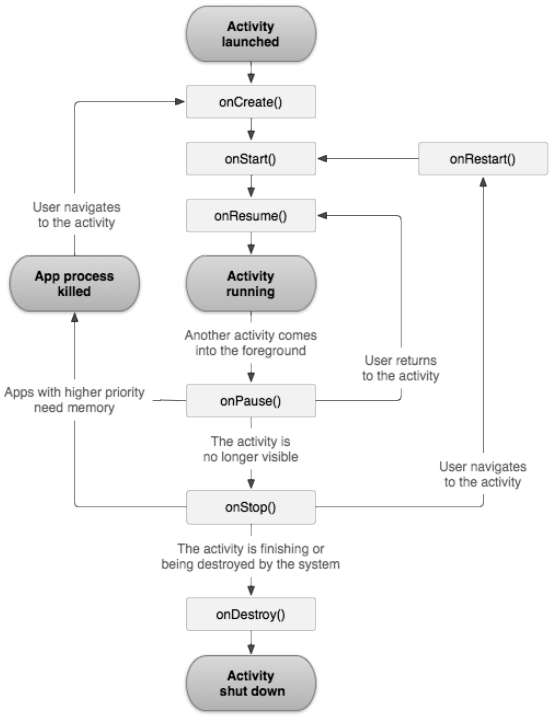

Fig. 1: The Android activity lifecycle. The ovals represent Android system calls, and the rectangles represent callback functions defined within an app. (Image taken from Android Developer Website[22].)

3) Control Flow Graph Representation: In a control flow graph, each node represents a single program statement, and each edge represents an allowed transition between statements (for example, an edge could represent direct transition to the next sequential instruction, or a branch such as a conditional jump). FlowDroid operates on an intermediate Java representation called Jimple [13], which is a typed three-address representation that sits between high-level Java and Android's native Dalvik virtual machine instructions in terms of complexity. An example of a Jimple-based control flow graph is shown in Figure 2.

Thus in the FlowDroid control flow graph, each node corresponds to a single Jimple statement, and each edge is one of four types [13]:

- Normal Edge - represents straight-line control flow from one statement to the next within the same method.
- Call Edge - represents a call from a node in one method

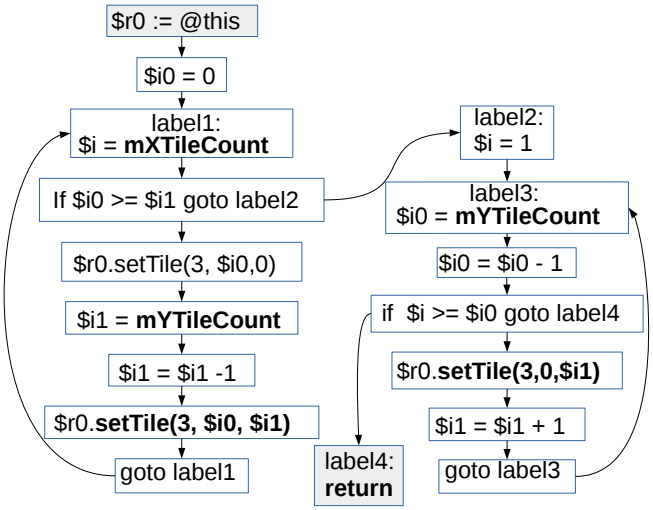

Fig. 2: An example of a portion of a Jimple-based interprocedural control flow graph for the classic mobile phone game "Snake". For simplicity, only the subgraph corresponding to the `updateWalls()` method is shown here. Call sites (which correspond to edges into other methods) are shown in bold.

to the start node of a different method.

- Return Edge - represents a return from the exit node of a method back to the next node in the calling method.
- Call-to-Return Edge - represents an edge connecting a call node and a return node. Clearly program execution does not proceed directly from the call node to the return node, however it is necessary to include this edge so that tainted state which is not passed to the call method is preserved after the return.

Note that each of these types corresponds to one of the transfer function types defined in Section V-A above.

4) Modifications for Path Length Analysis: FlowDroid keeps track of all tainted access paths, and only outputs those which terminate at a pre-defined sink [13]. In our analysis, we wish to capture the lengths of each tainted access path, in addition to the path origin. Additionally, we wish to examine the paths for *all* tainted variables, regardless of whether their propagation terminates in a sink. FlowDroid provides the `taintPropagationHandler` interface, which defines methods for a handler function which is invoked each time a taint is propagated [24]. We define the classes `Soot.jimple.infoflow.sgp.SGPHandlerLite` and `Soot.jimple.infoflow.sgp.SGPHandlerVerbose` which implement `taintPropagationHandler`. (These handlers differ only in the level of detail provided in the output; `SGPHandlerLite` provides path length output only, `SGPHandlerVerbose` gives path length along with flow fact details and taint source context for each propagation step). By setting these functions as callback methods at the outset of the data flow analysis, we are able to intercept incoming flow fact F, current statement s and outgoing set of flow facts $\mathcal{F}_{out}$ each time a taint is propagated. To give additional context to the paths measured, we also store the so-called source context (the API call which originated the given taint). In addition, we modify FlowDroid's native definition of the data flow fact class to include a unique identifier. Then at each invocation of our handler function, the following logic is executed:

- If F has not been seen before, initialize $F.\texttt{length} = 0$. Store original source of F_{in}.
- For each flow fact $f \in \mathcal{F}_{out}$:
  - $f.\texttt{length} = F.\texttt{length} + 1$
  - Store source context information for f.

Note that source context is only stored explicitly in the first flow fact generated after input from a source. Therefore in order to ensure that all captured paths retain context information, we recursively examine f's predecessor flow facts where necessary to recover original source context.

As discussed above, at this point we are characterizing spread relative to size *only*. We do not capture any information about what operations take place along a long tainted path. Recall that in a best-practice scenario, receipt of untrusted input should be immediately followed by validation and population of a strongly typed native data structure. Further computation would then take place using the native data structure only, never the raw input. The worst-practice scenario would involve use of raw input throughout the code with limited or no validation. Due to the manner in which taint propagation is defined in FlowDroid, our analysis considers an object X to be tainted if any subfield of X is tainted [13]. Thus, our analysis does not capture the good-practice case in which validated data is read into a native object - a long path of length l which starts with validation and native structure population is treated the same as a path of length l consisting exclusively of propagations of raw, unvalidated input. Automated analysis of what operations are actually being performed at each stage in the control flow graph is beyond the scope of this first stage analysis. However, our path length output serves as a well-defined starting point for this future analysis.

E. Path Length Normalization

In order to effectively compare tainted path lengths between applications, it is necessary to normalize the path lengths by some factor representative of the size and complexity of the control flow graph being analyzed. For example: a tainted path of length 90 in an app whose control flow graph has a total diameter of 100 should be considered to be more significant than a tainted path of length 90 in an app whose control flow graph has a diameter of 1000, as the former represents a taint which propagates much farther into the application.

The most correct strategy for normalization would be to calculate the distribution of path lengths, and use this to establish a normalization factor based on path length mean and standard deviation. Due to time constraints, for the purpose of this project we are taking a more naïve approach and normalizing

by an upper-bound on graph diameter[2]. Due to the nature of the Android life cycle and the fact that lifecycle methods can be invoked at any time and in any order, FlowDroid does not store a single data structure containing all nodes and edges [25]. The taint propagation engine's exploration of possible paths is more subtle than a simple iteration over a single graph. However, FlowDroid does provide well defined methods for extracting the subgraphs corresponding to individual methods, as well as an enumeration of all methods that are accessible. Thus we define our diameter upper bound D as the sum of diameters of each method subgraph, as follows:

$$\text{diameter} \leq D = \sum_{m \in \mathcal{M}} d_m$$

where $\mathcal{M}$ is the set of all accessible methods and d_m is the diameter of method m.

This approximation is a valid upper bound (albeit not a tight one), since the "worst case" longest path through the app would occur if every method were accessible from every other method, and within each method the control flow with the longest path was taken. The length of such a path would then be the sum of diameters of each method. While this calculation is not a perfect solution, it provides adequate accuracy for our first "rough cut" analysis. The diameter of each subgraph is calculated using the Floyd-Warshall shortest path algorithm.

1) Application Data Set: In order to study as large a cross-section of the app market as possible, we selected 55 applications from 15 of the major categories available on the Google Play store. It must be noted that due to regional restrictions, we were not able to download some apps out of particular categories (banking and media apps in particular often require the user's device to be registered in a specific country). FlowDroid's analysis is very computationally expensive, requiring up to 100 GB of RAM to analyze applications of significant complexity. Lacking the hardware and time for analysis on this scale, it was necessary for us to restrict our analysis to applications under a certain size. Preliminary testing indicated that using a machine with 32 GB of RAM, it was (usually) possible to analyze an application with total `.apk` size less than 10 MB in approximately 20 min. However, since `.apk` size does not necessarily increase linearly with control flow graph complexity, it was still necessary to discard some test applications on a case-by-case basis when out of memory errors were encountered. For reasons of runtime efficiency, we restricted the final test dataset to apps under 2 MB in size. Unfortunately, this size limitation meant that we were not able to sample apps from all categories - see the Limitations section for details. The number of apps sampled from each category is detailed in Table I. Although we do not have a large enough sample set to draw any significant conclusions about behaviour of apps from particular categories, we include the details of the categories used to underscore the fact that our sample covers a wide range of application types.

[2]Where diameter is defined as the longest shortest-path between any 2 nodes in the graph

Category	Number of Apps Analyzed
Books & Reference	4
Comics	7
Entertainment	1
Finance	5
Library & Demo	5
Live Wallpaper	5
Medical	2
Music	2
Personalization	1
Photo	1
Social	6
Tools	3
Transportation	3
Travel	4
Weather	4

TABLE I: Sample Applications from 15 Categories

F. A Note on Scope

It should be noted that while this work focuses on the analysis of Android applications, our static control flow graph-based methodology is not Android-specific. Our graph-based definition of the three characteristics of a shotgun parser can be applied to any code that uses data from an external source. We choose to focus on the Android platform first for several reasons. The open availability of Android application binaries allows us to examine byte code directly. Furthermore, choosing Android enables us to leverage existing tools (namely FlowDroid) for control flow graph construction. Additionally, as discussed in the Introduction, the popularity of the Android operating system in combination with the prevalence of third party libraries in Android app development mean that a single bug can impact huge numbers of users. As discussed by Patterson et al. in [20] and [21], the shotgun parser anti-pattern can cause significant security vulnerabilities, and as such there is value in exploring the extent to which this anti-pattern is present in a very widely-used mobile platform. A logical next step would be the application of our analysis technique to

other software ecosystems. However, analysis of code other than Android applications is beyond the scope of this paper.

VI. Application Analysis Results

We ran our analysis tool on 55 applications selected from 15 of Google Play's 25 categories. For each application, the length of each tainted path was recorded and normalized as described above. The results of our analysis are presented in the following sections.

A. Path Length Analysis For Individual Applications

We first discuss tainted path lengths on a per-app basis. Since we are interested in how many tainted paths of each length occur, we generate a histogram for each application. In each plot, we visualize the frequency of each observed (normalized)[3] path length for the given application. Binning for each histogram was calculated using the typical rule-of-thumb for binning: $number\ of\ bins = \sqrt{number\ of\ samples}$. (Note that the axes for histograms of different applications are not necessarily the same!)

The absolute number of tainted paths varies widely by application, which is as expected given the diversity in application size and complexity. Maximum path length varies by app as well, although it is worth noting that the majority of paths cover less than 10% of their respective graph diameters. This will be discussed in detail in a subsequent section.

Despite these variations, it is interesting to note that for most apps analyzed, the overall shape of the path length distribution is similar. Figures 3 and 4 are representative of the typical distribution shape. In general, tainted path lengths tend to follow an inverse power log-like shape, with the majority of tainted paths being very short. Such a distribution is promising from the standpoint of avoiding the large-spread-relative-to-size anti-pattern, as this distribution indicates that the majority of tainted input propagates a very short distance.

We are also interested in deviations from the typical distribution shape. Such deviations can serve as a starting point for analysis of the second and third hallmarks of the shotgun parser, namely large numbers of tainted variables and use before full recognition. We observed several such cases in our sample set. For example, rather than the inverse power log-like shape, the "Garfield Daily" comics app (shown in Figure 5) has an almost trimodal shape. Perhaps even more interesting is the "Canadian Tire Money Tracker" app (Figure 6, which has a large cluster of very short tainted paths with lengths between 0.1 and 0.2.

Perhaps the best candidate for further analysis is the "Open Comic Reader" application. The large majority of this app's tainted paths are less than 20% of the total diameter, and so upon first inspection it might appear that this app follows the typical inverse power log shape, as shown in Figure 7.

Examination of the full data range tells another story. As can be seen in Figure 8, there are several instances of very long paths, including one group of paths whose lengths exceed 80% of the graph diameter. Furthermore, these are not single paths, but clusters of paths which occur with enough frequency to be observable even in a sample set with path length counts of up to 90000. These are certainly instances of large spread relative to size.

What operations are taking place along these very long paths? What pieces of state are involved? These questions will be the subject of our future work to construct an identifier for the second and third properties of the shotgun parser. The Open Comic Reader case demonstrates how our current work serves as a foundation for future analysis - we have identified a "likely candidate", and can now focus further analysis on a small subset of paths in a particular application.

[3]Since we normalize by (upper bound of) diameter, this means that each length is a value between 0 and 1.

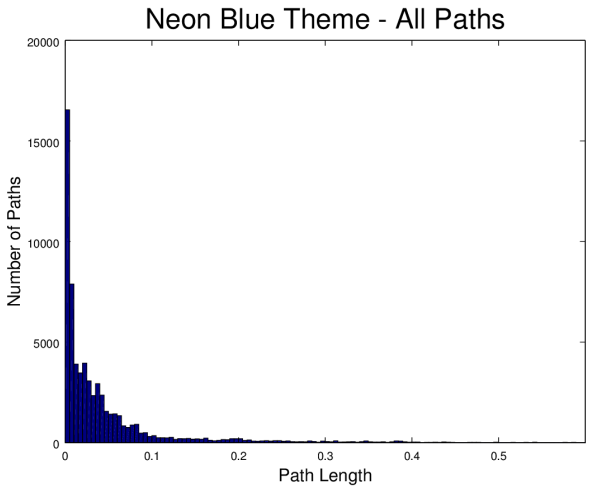

Fig. 3: Histogram of tainted path length occurrences for the Neon Blue Theme wallpaper app, displaying common inverse power log-like distribution

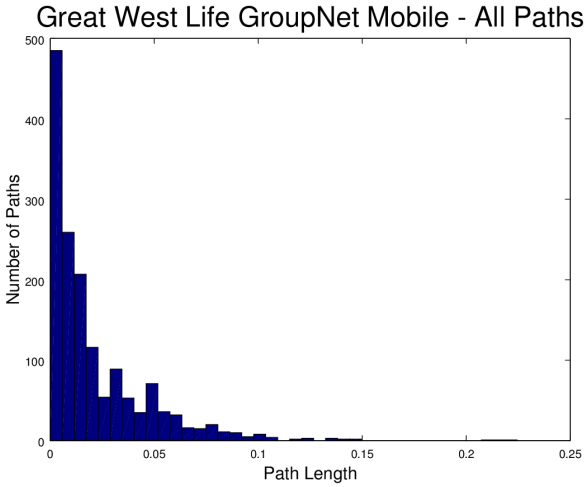

Fig. 4: Histogram of tainted path length occurrences for the Great West Life Insurance GroupNet app, also displaying common inverse power log-like distribution

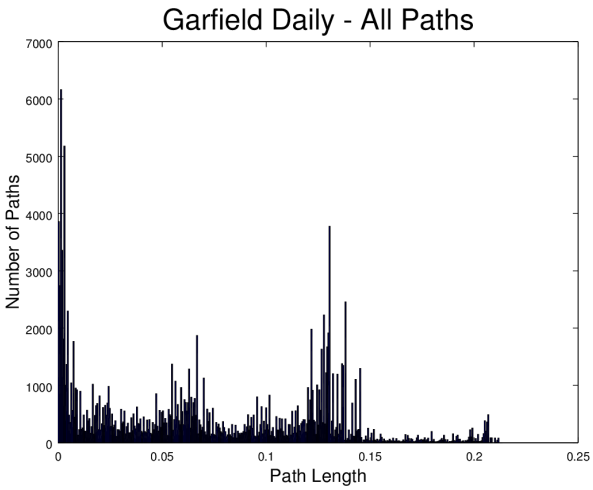

Fig. 5: Histogram of tainted path length occurrences for the Garfield Daily comic reader. This shape of the distribution for this application deviates from the typical shape, which makes this app of interest for further study.

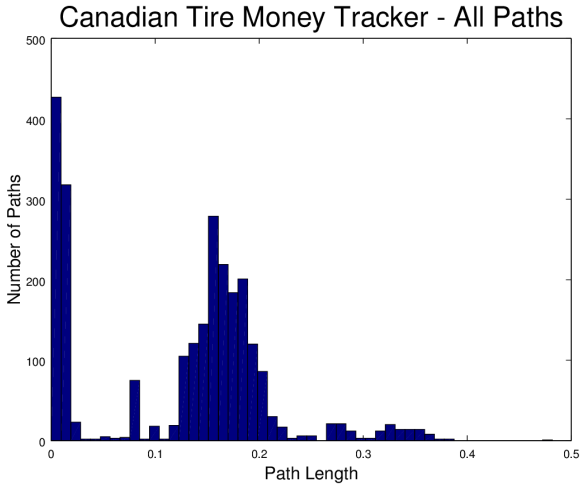

Fig. 6: Histogram of tainted path length occurrences for the Canadian Tire Money Tracker app. This shape of the distribution for this application deviates from the typical shape, which makes this app of interest for further study, particularly since there is a large cluster of tainted paths which span 15-20% of the total control flow graph diameter.

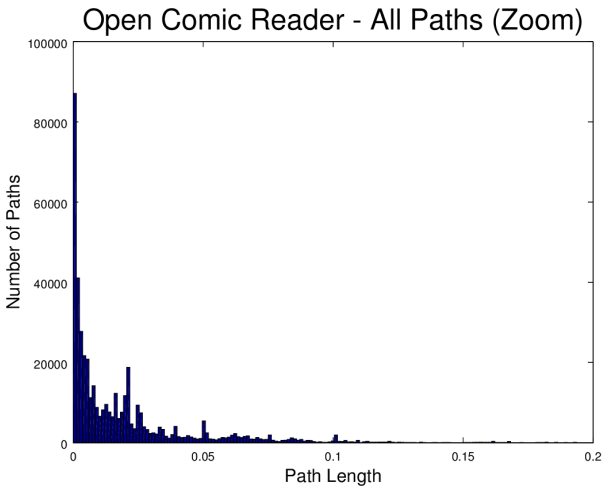

Fig. 7: Histogram of tainted path length occurrences for the Open Comic Reader app. For path lengths between 0 and 0.2, this application appears to display the typical inverse power log shape seen in other applications.

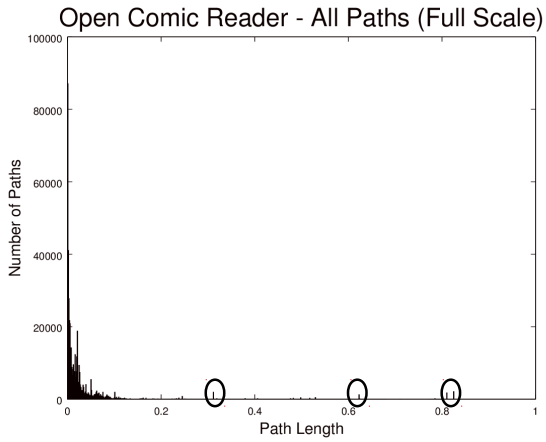

Fig. 8: Histogram of tainted path length occurrences for the Open Comic Reader app, shown at full scale. The circled paths span a significant proportion of the control flow graph and thus are of interest for further study.

The *Library & Demo* category deserves a special mention. Referring to the full set of histograms (available on GitHub - see Appendix A), it can be seen that the tainted paths in applications from this category tend to be very few in number. We theorize that this variation results from the nature of the *Library& Demo* classification - this category appears to be reserved for small apps which either demo a specific feature, or which work as an extension to another, larger application. For example, the “Aviary Effects: Classic” app (shown in Figure 13 in Appendix A) is a plugin for the Aviary Photo Editor.

Thus, since most functionality is handled by the primary app, these small plugin apps have simple control flow graphs and short path lengths. However, analysis of a much larger sample set would be required to draw any general conclusions about the general behaviour of apps in *Library & Demo*.

B. Source Type Analysis

Each tainted path originates at a source, and so it is of interest to examine the relationship of source type to tainted path length. To this end, we combined the path length data for all 56 applications in our data set and calculated the minimum, maximum and average path length for each of the 1080 unique source types observed.

Figure 9 displays the minimum, maximum and average path lengths for the 20 sources with the longest average tainted path length. (We visualize only the top 20 sources for the sake of clarity). The taint source class and method call are detailed on the left hand side of the plot. It can be seen that the source corresponding to the longest average path is `android.content.Context.getContext()`, called from within the `com.aviary.android.feather.plugins.filters.library.BaseEffectsContentProvier` library. Indeed, examining the histogram for the Aviary Effects: Classic app (Figure 13) reveals that the small number of paths are relatively evenly distributed between 0 and 50% of the graph diameter, resulting in a high average path length. It is also interesting to note that many of the sources have a minimum path length of zero, which indicates that a call to the source was detected but that data from this source was never propagated.

In Figure 10 we display the minimum, maximum and average path lengths for the 20 sources with the longest maximum tainted path length. This is an interesting figure in the context of building a foundation for further analysis, as it highlights those sources with a high probability of being associated with shotgun parser indicators. We see here that the worst offender is the method `getParcelable(String string)`, which returns the Parcel value associated with the argument string [26]. An Android Parcel is an object designed for high-performance inter-process communication (IPC) [27]. Thus, the most common type of input data being propagated in our longest tainted paths is an object designed to transmit messages from one process to another. This presents a definite security concern, since malicious applications can co-exist along with benign applications on the same device, and as such any inter-process communication should be handled carefully. In particular, this situation is concerning in the context of the "confused-deputy" style attack, wherein a malicious process delegates a task for which it does not have privilege to a more privileged, naïve process [23]. This case is another excellent proof-of-concept for our tool, as we have identified a possible source of concern and can now focus further analysis on *how* data from this untrusted source is handled - i.e. whether the input is fully recognized before being used.

C. Average Tainted Path Length for All Applications

Finally, we examine the average tainted path length for each application analyzed. This is shown in Figure 11. We note that all average path lengths are less than 30% of their graph diameter, and the majority of tainted paths are less than 5% of their graph diameter. This appears to be a promising result indicating a relatively low overall prevalence of the spread relative to size anti-pattern.

VII. Future Work and Challenges

A. Limitations

Several limitations of our approach must be noted.

- Data Set Size and Diversity
  Due to time constraints and hardware limitations, our sample set was relatively small and restricted to very small apps. In particular, we were not able to assess any apps from the extensive Games category, as popular game applications were too large to be analyzed by our tool in a reasonable timeframe. In order to fully characterize the spread relative to size characteristic, it would be necessary to have a much larger sample including applications of various sizes and levels of complexity. Future work will include work to improve the memory requirements of our analysis tool.
- Version Specific Android Source and Sink List
  As discussed in the *Methodology* section, we use the output of the SuSi tool to specify sources and sinks for FlowDroid. SuSi's output depends on the specific version of the Android API being analyzed, and it has been observed that there are differences between API versions [15]. Therefore to be most complete, it would be necessary to run SuSi on each version of the Android API we wish to support, and then use the appropriate source/sink list for the API version each app was developed with (version information is readily available in each application manifest file so this aspect would not be a challenge to implement).

B. Verification

In this work we have proposed a theoretical definition for a shotgun parser and have conducted an initial investigation of the spread relative to size property. An important next step will be to more thoroughly study the practical manifestations of this property in source code. The following approaches are suggested:

- Study of actual shotgun parser-based vulnerabilities
  An ideal test of our tool would be to assess its performance on applications which contain known vulnerabilities stemming from unprincipled input handling. In particular, it would be valuable to confirm the correlation between very long input paths and shotgun parsing. This analysis will necessitate some extensions to our tool. Since FlowDroid operates by creating a dummy main method which invokes all possible life cycle methods in all possible orders, it will be necessary to identify which

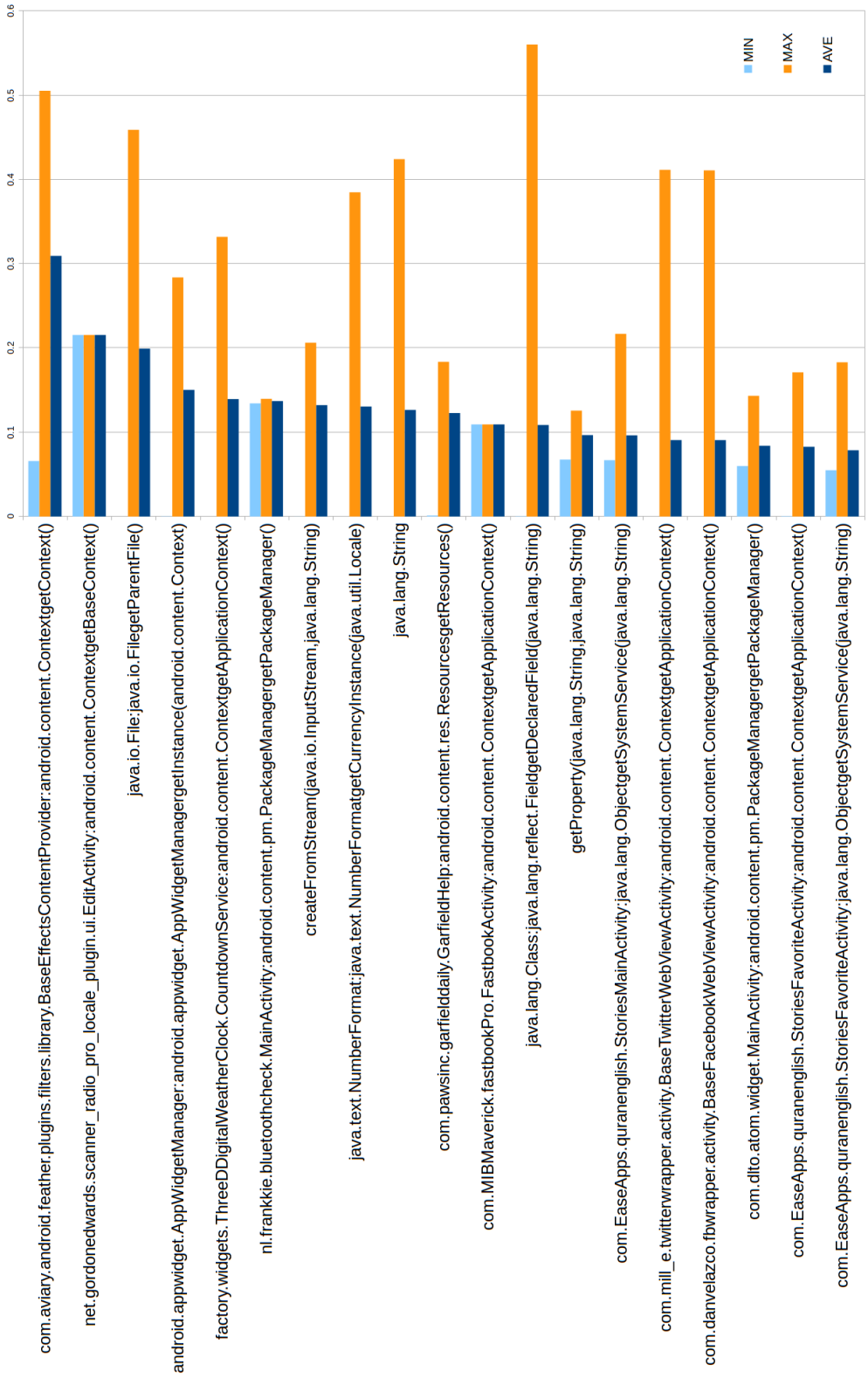

Fig. 9: Data for the 20 source types with the longest average tainted path length.

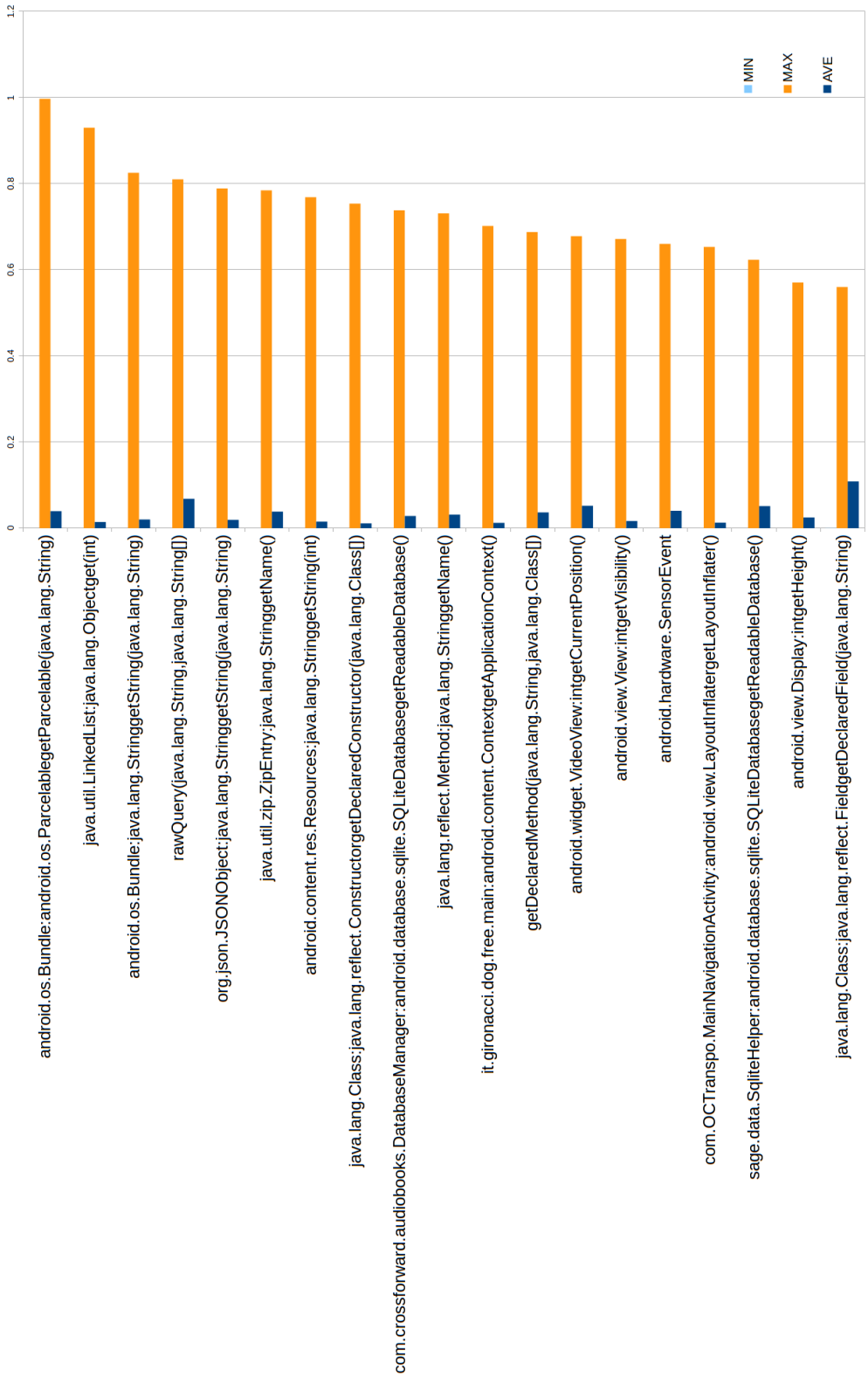

Fig. 10: Data for the 20 sources with the longest maximum tainted path length.

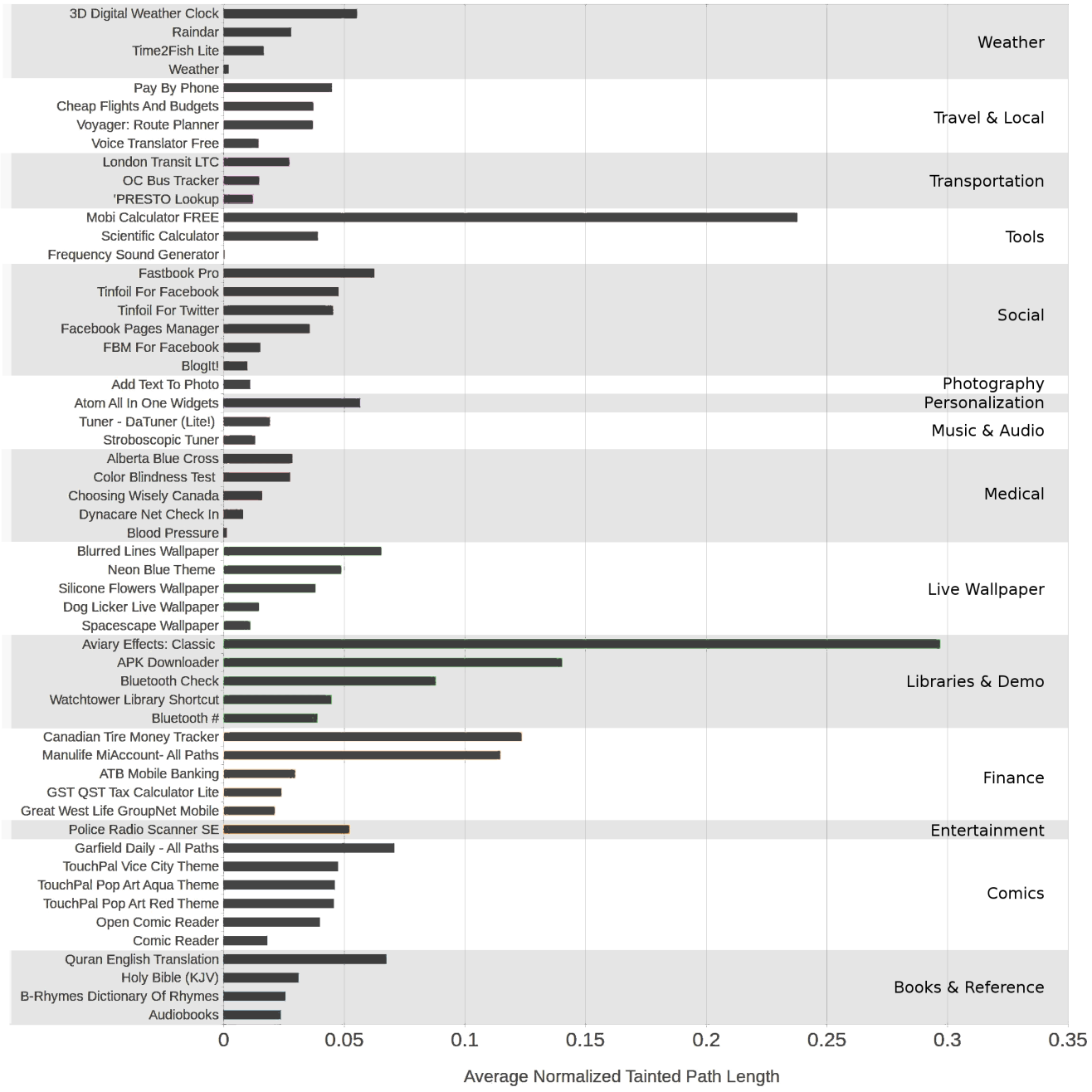

Fig. 11: Average (normalized) tainted path length for each application in the data set.

specific ordering corresponds to each long tainted path, so that the corresponding execution path through the source code can be precisely traced. However, selection of a test set of applications for this analysis may prove challenging. It will be necessary to examine the application source code to confirm that the known vulnerability did in fact result from improper handling of input data, and while Android application binaries are readily available, application source code is not always so.

- Input fuzzing
  It will also be valuable to identify the nature of the input data involved in very long propagations. This can be achieved by fuzzing the application and tracking which combinations of input cause the very long access paths to be exercised.

C. Second and Third Shotgun Parser Properties

As described in the Methodology section, this work focused on identification of the first of three proposed properties of a shotgun parser. Thus, the natural extension to this work is implementation of automated detection of the second and third properties, namely number of input variables involved in each tainted path and use before full recognition. Extending our tool to capture the number of tainted variables involved in a given path will enable us to assign edge weights to each path, with the weight corresponding to the number of tainted variables involved. This step will allow us to further narrow down areas of concern for further analysis.

The key step will be to address the use before full recognition property. That is, we must characterize the nature of the operations taking place along each long tainted path and assess whether or not these operations constitute "full recognition" of the input data. In order to determine whether input data is being fully recognized before use, we must answer the following questions:

- Does there exist code which constitutes a parser?
- Is this code used to *completely* parse the input data before that data is used in any way?
- Is the nature of the parser appropriate for the language complexity of the input data? (For example, it is inappropriate to attempt to validate a context-free input language using a regular grammar [20]).

Automated verification of these conditions is a problem of significant complexity and will constitute the majority of the future work on this project.

We can gain some insight into the use before full recognition problem by examining the sequence of memory read and write operations that occur for given pieces of program state involved in long propagations. As a first step toward addressing this property, we propose to extend our existing framework to capture the Jimple statement associated with each taint propagation, and thus reconstruct the sequence of Jimple statements corresponding to each identified path through the control flow graph. By examining these sequences, we can extract the read and write events associated with each component variable over the course of a taint propagation.

By characterizing sequences of read and write operations in this manner, we can better understand the nature of the operations occurring along identified long paths and assess whether input data is handled in a principled manner. Furthermore, through empirical examination of such read/write sequences extracted from known shotgun parsers, we can begin to construct a heuristic describing "good" and "bad" behaviour in terms of input parsing. We can also begin to set a policy for each Android data type (an example of such a rule might be "each field of object type x must be written to before any field of object type x is read"). Of course, such rules would only encompass behaviours related to loading and storing memory operations - we do not propose to offer a complete set of rules for all possible operations on all possible objects.

It must be noted that there are still several limitations inherent to this approach. Jimple is an intermediate representation, and while it does accurately represent the intent of the original bytecode, it does not necessarily provide a one-to-one representation of the memory events which occur at a bytecode level [28]. Secondly, the Jimple transformation represents stack positions with additional *local* variables, and local variable naming is unique within methods only [28]. This means that at best, we can describe the read/write events on a per-variable, per-method basis, which limits the power of this technique. Finally, since our analysis is purely static, we are not able to extract the nature of the memory events that occur at runtime (this is of course a challenge associated with the static analysis of any system, and is not unique to our system or the Android platform in general).

While the approach outlined above will form the basis of our future work in the Android domain, we also propose to port the analysis techniques described in this paper to an x86 framework. With this ported system, we can test the validity of our shotgun parser definition through analysis of known shotgun parser samples.

VIII. Conclusion

In this work, we addressed the problem of unprincipled handling of input data in Android applications from a LangSec approach. In particular, we focused on identification of the shotgun parser security anti-pattern. After establishing three characteristics of a shotgun parser, we turned our attention to identifying the first of these hallmarks: spread of tainted input data relative to application control flow graph size. By modifying the FlowDroid tool for static taint analysis, we were able to develop a tool which measures the propagation path length of tainted data originating at each input source in a target Android application. After analyzing a set of 55 applications to quantify the spread of tainted data relative to interprocedural control flow graph diameter, we were able to draw several conclusions. Although there is a significant variation between apps in terms of the number of tainted paths observed, we noted that the shape of the distribution of path lengths was very similar between applications, even those from disparate categories. In most applications, the majority of tainted path lengths were short, on average less than 5% of

the overall graph diameter. However, we also found instances of very long tainted paths, both in individual applications and associated with particular input sources. These results are promising for several reasons. The fact that the majority of average tainted path lengths are less than 5% of their respective graph diameters in length suggests that the prevalence of the spread relative to size anti-pattern is not overwhelming. At the same time, our tool was successful in identifying instances of tainted paths which *were* long compared to their respective diameters. In addition to being useful information in and of itself, identification of these long paths allows us to classify likely shotgun parser candidates. We can then focus on these candidates in the next phase of analysis, which will be concerned with identification of the second and third of the shotgun parser characteristics.

IX. Acknowledgements

The authors gratefully acknowledge Steven Arzt at the European Centre for Security and Privacy by Design for his ongoing assistance with technical questions about FlowDroid via the Soot mailing list.

References

[1] I. D. C. (IDC), Online article, 2015. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] J. Drake, "Stagefright: Scary code in the heart of android," in *Black Hat Briefings*, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf

[3] S. Bratus, T. Darley, M. E. Locasto, and M. L. Patterson, "Langsec: Recognition, validation, and compositional correctness for real world security," USENIX Security BoF Handout, 2013. [Online]. Available: http://langsec.org/bof-handout.pdf

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[7] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," in *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[8] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic Execution for Dalvik Bytecode," Department of Computer Science, University of Maryland, College Park, Tech. Rep. CS-TR-5022, July 2012.

[9] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[10] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 611–622. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516689

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[13] C. Fritz, "Flowdroid: A precise and scalable data flow analysis for android," Master's thesis, Technische Universitat Darmstadt, 2013.

[14] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," EC SPRIDE, Tech. Rep. TUD-CS-2013-0113, May 2013. [Online]. Available: http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf

[15] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," EC SPRIDE, Tech. Rep. TUD-CS-2013-0114, May 2013.

[16] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09. New York, NY, USA: ACM, 2009, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1554339.1554341

[17] M. Dam, G. Le Guernic, and A. Lundblad, "Treedroid: A tree automaton based approach to enforcing data processing policies," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 894–905. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382290

[18] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 359–368. [Online]. Available: http://doi.acm.org/10.1145/2420950.2421003

[19] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 590–604. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.44

[20] M. L. Patterson, S. Bratus, and D. Hirsch, "Shotgun parsers in the cross-hairs," Presented at Brucon 2012, 2012. [Online]. Available: https://www.youtube.com/watch?v=mLc0cwlVe84

[21] ——, "From 'shotgun parsers' to better software stacks," Presented at Shmoocon 2013, 2013. [Online]. Available: https://www.youtube.com/watch?v=XVZrmp5MAas

[22] A. D. Guide, Online article, 2015. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html

[23] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028089

[24] S. Arzt, personal communication via Soot mailing list, 09 2015. [Online]. Available: https://mailman.cs.mcgill.ca/pipermail/soot-list/2015-September/008260.html

[25] ——, personal communication via Soot mailing list, 11 2015. [Online]. Available: https://mailman.cs.mcgill.ca/pipermail/soot-list/2015-November/008305.html

[26] A. D. Guide, Online article, 2015. [Online]. Available: http://developer.android.com/reference/android/os/Bundle.html

[27] ——, Online article, 2015. [Online]. Available: http://developer.android.com/reference/android/os/Parcel.html

[28] R. Vallee-Rai, "Soot: A java bytecode optimization framework," Master's thesis, School Of Computer Science, McGill University, 2000.

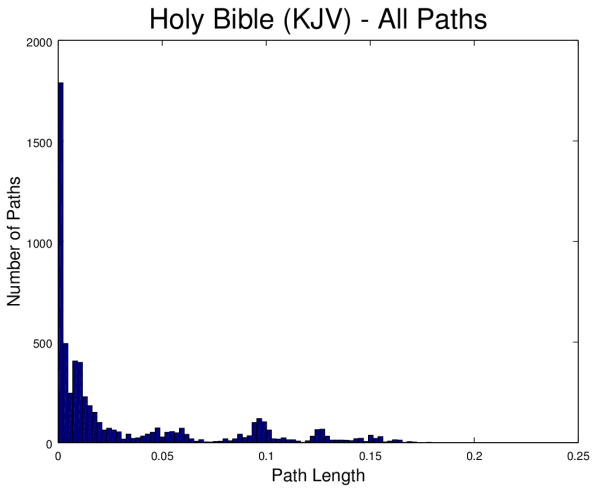

Fig. 12: Category: Books & Reference. Also displays some variation from the typical distribution pattern.

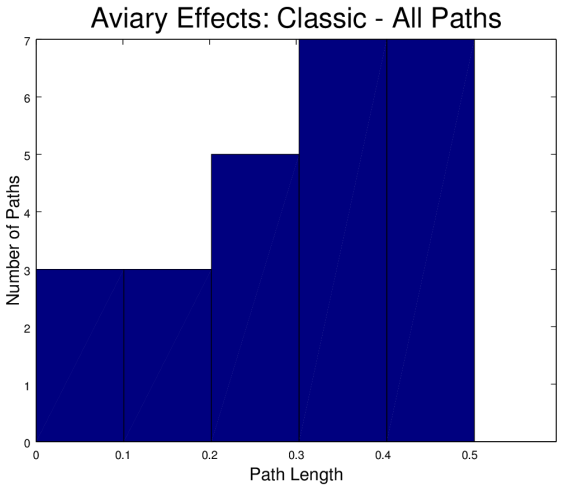

Fig. 13: Category: Library & Demo

Appendix

In this section, we present some additional path length histograms of interest. Each histogram displays the frequency with which different (normalized) path lengths occur. Note that the axes are *not* the same for each plot - each axis was chosen to best represent the data range being displayed.

The path length histograms for all 55 applications tested can be viewed on our GitHub site at https://github.com/sgpSearch/ShotgunParsersInAndroidApplications.

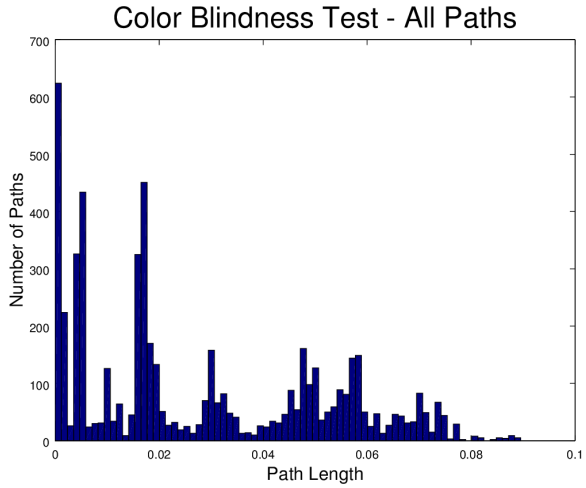

Fig. 14: Category: Medical. Appears to deviate from the typical distribution, but note also that there are no path lengths greater than 0.1. Therefore overall spread relative to size is still low.

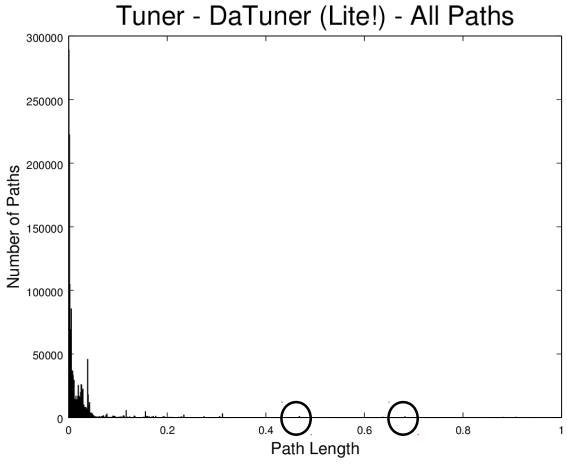

Fig. 15: Category: Music & Audio. Note the small circled clusters of long paths at 0.5 and 0.7.

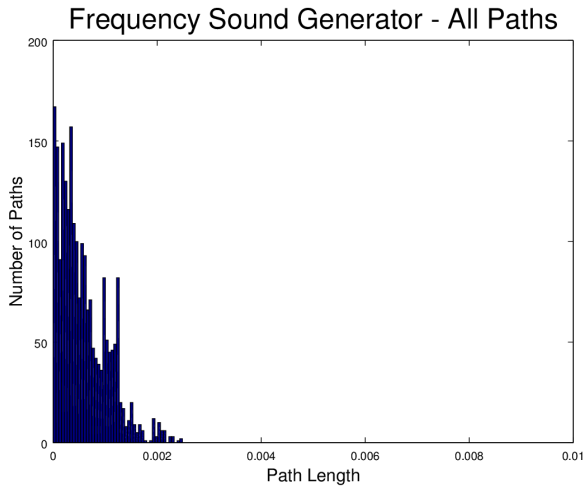

Fig. 16: Category: Tools. This application has a very small spread relative to size as well as a comparatively small number of paths.

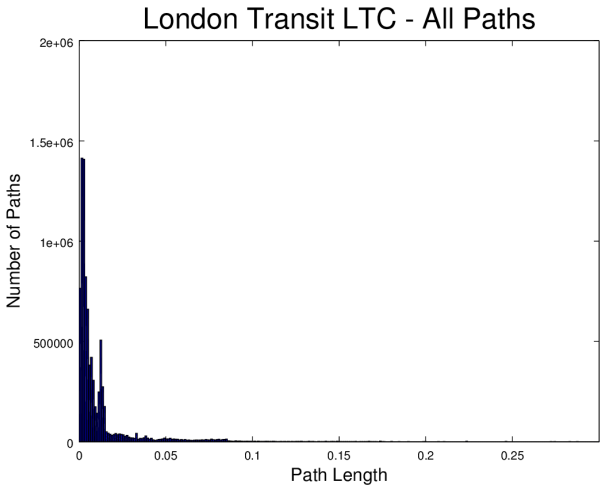

Fig. 17: Category: Transportation. This app deserves special mention for having the largest number of paths of any application surveyed.

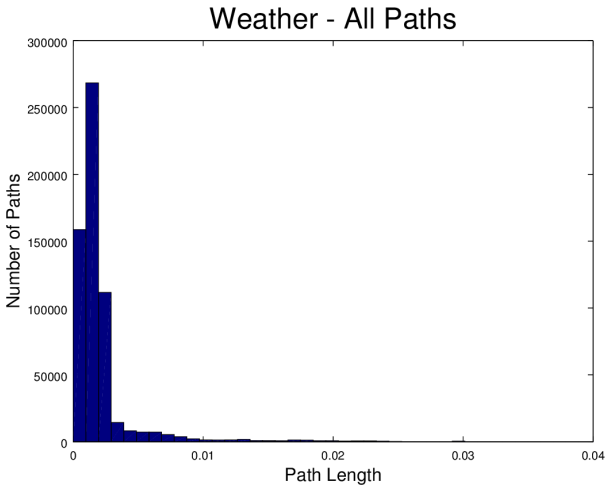

Fig. 18: Category: Weather. Another special mention - we observed a very large number of tainted paths, however the longest path is only 0.04 of the total diameter indicating a very small spread relative to size.