# CFL Shortest Path for Minimum Model Problems

Osbert Bastani

October 6, 2012

## 1 Introduction

The Android platform is experiencing a proliferation of malware largely analagous to the early proliferation of malware on Windows systems. Even as new techniques are developed to detect malware applications, more sophisticated malware is developed to circumvent these protections. Ideally we would be able to provide provable guarantees about the safety of a given Android application. One promising approach is through the use of static analysis.

Static analysis techniques can be used to determine the flow of information through a program. A large number of malware applications are characterized by malicious flows of information, either from or to the Android device. For example:

1. a malware application that reads and sends stored passwords to a malicious web server has a flow from the stored passwords to the internet,
2. a malware application that displays advertisements for web pages, possibly scams, has a flow from the internet to the device display,
3. a malware application that calls premium phone numbers without the user's permission has a flow from the device to the dialed phone number.

Detecting android malware would be greatly improved with a reliable understanding of information flow through the application. Unfortunately, performing static analyses on Android applications is greatly complicated by use of the extensive Android SDK, primarily because of the following problems:

1. the sheer size of the SDK means that static analysis does not scale well,
2. the Android SDK sometimes utilizes C or C++ libraries, which are more difficult to analyze.

One solution is to replace the Android SDK with models that capture the relevant properties of the methods. For example, when analyzing information flow, the models should describe flow between arguments and to the return values.

Unfortunately, there is currently no way to automatically generate the models, so they must be provided by human auditors. Due to the number of methods in the Android SDK, and the evolving nature of the SDK, it is impractical for a small team of auditors to generate all the models by hand. Oftentimes, when trying to answer some query $q$ about the program, it suffices to implement only a small subset of missing models. Hence we have the following two problems:

**Problem 1.1.** Find the smallest set of models that must be implemented to have a good chance (according to some heuristic) to prove $q$ true.

**Problem 1.2.** Find the smallest set of models that must be implemented to have a good chance (according to some heuristic) to prove $q$ false.

The goal of this paper is to formalize and solve this problem for the case of finding information flow. For now, we will focus on Problem 1.1.

## 2 Background

Here, we review the traditional flows-to graph for a program. Given a program $P$, let $F$ be the set of fields in $P$, and let $L$ be the context-free language generated by the following grammar:

1. a set of terminals

$$\Sigma = \{\text{alias}\} \cup \{\text{store}_f, \text{load}_f\}_{f\in F},$$

2. the terminal symbol $T$,
3. the set of productions

$$\Pi = \{T \to \epsilon, T \to \text{alias}, T \to TT\} \cup \{T \to \text{load}_f T \text{store}_f\}_{f \in F},$$

where $\epsilon$ is the empty string.

Then the **flows-to graph** $G$ for the program $P$ consists of the following data:

1. a set of vertices $V$, one for each reference in $P$,
2. a set of edges $E \subset V \times V \times \Sigma$.

We think of $s \in \Sigma$ as a label on the edge $(x, y) \in V \times V$, so we write such an edge as $x \xrightarrow{s} y$. Note however that there can be multiple edges $(x, y)$, each with a different label. Given references $x$ and $y$, we include the following edges:

1. if there is an expression $y = x$, include the edge $x \xrightarrow{\text{alias}} y$,
2. if there is an expression $y.f = x$, include the edge $x \xrightarrow{\text{store}_f} y$,
3. if there is an expression $y = x.f$, include the edge $x \xrightarrow{\text{load}_f} y$.

Then we have the following:

**Proposition 2.1.** Given a Java program with flows-to graph $G$, information flows from reference $x$ to reference $y$ only if there is a path $(e_1, ..., e_n)$ from $x$ to $y$ such that the sequence $s_{e_1}...s_{e_n} \in L$, where $s_e \in \Sigma$ denotes the label on edge $e$. We call such a path a **CFL path**.

This useful tool for capturing information flow through a program will be the basis for our analysis. We will augment the flows-to graph to handle the case of missing edges in the program, and then give an algorithm to find the minimum set of models according to some heuristic. Now we can state the information flow problem. Given a pair of references (source,sink), we would like to determine whether information can flow from source to sink. More precisely:

**Problem 2.2.** Given a program $P$ with flows-to graph $G = (V, E)$, and vertices $v_{\text{source}}, v_{\text{sink}} \in V$, determine whether there exists a CFL path from $v_{\text{source}}$ to $v_{\text{sink}}$.

When there are no missing models, this problem can be solved using a CFL reachability algorithm, as in [1]. We now turn to the issue of formulating this problem when the program has methods that are missing models.

## 3 Information Flow with Missing Models

We begin with a motivating example. Consider the following code:

```
$x \leftarrow$source
$y.f \leftarrow x$
$z \leftarrow$foo1($x$)
$w \leftarrow$foo2($y$)
sink$\leftarrow z$
sink$\leftarrow w$
```

The associated graph $G$ is given in Figure **??**. It is clear that the following definitions of foo1 will induce information flow from the source to the sink:

```
procedure FOO1(x)
    return $x$
end procedure
```

and similarly for foo2:

```
procedure FOO2(y)
    return $y.f$
end procedure
```

In the remainder of this section we will develop an algorithm to handle the set of "possible" edges that can occur when a model is added. Since the set of such edges is technically infinite, we will need to introduce new algorithmic tools in order to do so.

### 3.1 "Possible" Edges

In Figure **??**, we show the graph $G$ augmented with a pair of "possible" edges, represented by dashed arrows, that occur if foo1 and foo2 are defined as above. There are many other possible edges that can occur; the two shown happen to be the simplest edges that induce information flow. For example, we may have the following definition of foo2:

```
procedure FOO2(y):
    $t \leftarrow y.f$;
    return $t.g$;
end procedure
```

In this case, the flow would actually be $y \xrightarrow{\text{load}_f} t \xrightarrow{\text{load}_g} w$. To make this code valid, we would have to add a line such as $y.f.g$ =source to the program source code, but this is not important for the purposes of this discussion. In general, the implementation of a method may introduce new vertices into the graph $G$. In order to handle these in a general way, we can instead add a single edge of the form $y \xrightarrow{\text{load}_f\text{load}_g} w$ that implicitly assumes the existence of the vertex $t$.

Now we are close to being able to precisely define the set of "possible" edges in the augmented graph. For example, consider the following process:

```
procedure GETPOSSIBLEEDGES($P, \Lambda$)
    $E \leftarrow$new Set()
    for expression $y = foo(x_1, ..., x_n)$ in $P$ do
        if $foo$ does not have a model then
            // add edges from $x_i$ to $x_j$
            for $1 \leq i \leq n$, $1 \leq j \leq n$, $l \in \Lambda$ do
                $E$.add($x_i \xrightarrow{l} x_j$)
            end for
            // add edges from $x_i$ to $y$
            for $1 \leq i \leq n$, $l \in \Lambda$ do
                $E$.add($x_i \xrightarrow{l} y$)
            end for
        end if
    end for
    return $E$
end procedure
```

Here $\Lambda$ is the set of allowed labels for "possible" edges. Our initial choice for $\Lambda$ is $\Lambda = \Sigma^*$, i.e. the set of all possible strings of elements in $\Sigma$. There are two issues with this choice:

1. the set $\Lambda = \Sigma^*$ is infinite,
2. the number of possible $l \in \Sigma^m$ is exponential in $m$ (with base $\#\Sigma = \#F$), so the complexity blows up even if the length of $l$ is bounded, i.e. we choose $\Lambda = \Sigma^m$ for some $m$.

For now, we will solve the first issue by bounding the length of $l$ by $m$, so we focus for now on solving the second issue. First, it is clear that we can restrict the set of possible edges to a subset of $\Sigma^m$. Drawing again on our example, consider the edge $x \xrightarrow{\text{store}_f\text{load}_g} y$, which is equivalent to $x \xrightarrow{\text{store}_f} t \xrightarrow{\text{load}_g} y$. The points $x$ and $y$ are not connected in the context-free language sense since the store$_f$ and load$_g$ operations do not have matching fields. Any matching store and load operations must have matching fields.

Furthermore, consider the sequence $x \xrightarrow{\text{store}_f\text{load}_f} y$, which is equivalent to $x \xrightarrow{\text{store}_f} t \xrightarrow{\text{load}_f} y$. This indicates that the method foo1 would look like

```
procedure FOO1(x)
    $t.f \leftarrow x$
    return t.f
end procedure
```

But having $t.f$ is extraneous, and we could very well have just returned $x$ as before. Since we are only interested in the simplest possible models, we can delete the matching parenthesis. With some thought, we have the following proposition:

**Proposition 3.1.** The values of $l$ that we need to consider are of the form

$$\text{load}_{f_1}...\text{load}_{f_n}\text{store}_{f_{n+1}}...\text{store}_{f_{n+m}}.$$

However, while we have simplified the problem, there is still an exponential blowup in the number of edges that must be added. To fully solve the both issues, we will have to augment the language $L$ that we use.

## 3.2 Augmented Flows-To Graph

Here we redefine the language $L$ so that it can more flexibly handle edges representing missing models. The key idea is to note that along a "possible" edge, we can match any store and load opertions that occur before or after that edge. We would like to write an expression of the form

$$\text{load}_*...\text{load}_*\text{store}_*...\text{store}_*,$$

where the store$_*$ edge matches any terminal load$_f$ and load$_*$ matches any terminal store$_f$. More precisely, we define the context-free language $L$ to be language generated by the following grammar:

1. the set of terminals

$$\Sigma = \{\text{alias}, \text{store}_*, \text{load}_*\} \cup \{\text{store}_f, \text{load}_f\}_{f \in F},$$

2. the terminal symbol $T$,

3. the set of productions

$$\begin{aligned}\Pi = &\{T \to \epsilon, T \to \text{alias}, \\ &T \to TT, T \to \text{store}_* T \text{load}_*\} \\ &\cup \{T \to \text{store}_f T \text{load}_f, T \to \text{store}_f, \\ &T \to \text{store}_f T \text{load}_*, \\ &T \to \text{store}_* T \text{load}_f\}_{f \in F}.\end{aligned}$$

Furthermore, we define the set

$$\lambda_k = \begin{cases} \{\text{alias}\}, & \text{if } k = 0 \\ \{\text{store}_*, \text{load}_*\}^k, & \text{if } k \geq 1. \end{cases}$$

By Lemma 3.1, we need only consider values

$$l \in \Lambda = \bigcup_{k=0}^{\infty} \lambda_k = (\text{store}_*)^*(\text{load}_*)^*.$$

Here the $^*$ is the Kleene star operator. On some reflection, it is clear that an edge $x \xrightarrow{l \in \Lambda} y$ is equivalent to the set of edges

1. $x \xrightarrow{\epsilon} u \xrightarrow{\epsilon} v \xrightarrow{\epsilon} y$,

2. $u \xrightarrow{\text{store}_*} u$,

3. $v \xrightarrow{\text{load}_*} v$.

This is because the new construction allows arbitrarily many loops store$_*$ followed by arbitrarily many loops load$_*$. Hence we have the following simple redefinition of getPossibleEdges:

```
procedure GetPossibleEdges($P$)
    $E \leftarrow$new Set()
    for expression $y = foo(x_1, ..., x_n)$ in $P$ do
        if $foo$ does not have a model then
            // add edges from $x_i$ to $u_{x_i}$,
            // edges from $v_{x_i}$ to $x_i$ and $v_y$ to $y$,
            // and loops on $u_{x_i}$, $v_{x_i}$, and $v_y$
            for $1 \leq i \leq n$ do
                $E$.add($x_i \xrightarrow{\epsilon} u_{x_i}$)
                $E$.add($v_{x_i} \xrightarrow{\epsilon} x_i$)
                $E$.add($u_{x_i} \xrightarrow{\text{store}_*} u_{x_i}$)
                $E$.add($v_{x_i} \xrightarrow{\text{load}_*} v_{x_i}$)
            end for
            $E$.add($v_y \xrightarrow{\epsilon} y$)
            $E$.add($v_y \xrightarrow{\text{load}_*} y$)
            // add edges from $u_{x_i}$ to $v_{x_j}$ to $v_y$
            for $1 \leq i \leq n$, $1 \leq j \leq n$ do
                $E$.add($u_{x_i} \xrightarrow{\epsilon} v_{x_j}$)
            end for
            for $1 \leq i \leq n$ do
                $E$.add($u_{x_i} \xrightarrow{\epsilon} v_y$)
            end for
        end if
    end for
    return $E$
end procedure
```

Note here that we have included a further simplification, namely we require only one vertex $u_{x_i}$ and one vertex $v_{x_i}$ for each argument $x_i$ of a function call. This is acceptable since an edge going into $u_{x_i}$ can lead to any vertex $v_{x_j}$, so there is no need to have a distinct vertex for each flow $x_i \to x_j$.

The **augmented flows-to graph** $G$ is then the graph

1. a set of vertices $V$, one for each reference in $P$,

2. a set of edges $E \subset V \times V \times \Sigma$,

3. a set of "possible" edges $E' \subset V \times V \times \Sigma$.

The sets $V$ and $E$ are defined as before, and the set $E' = \text{getPossibleEdges}(P)$. Now that we have a formalized the notion of missing models in the flows-to graph, we can turn to formalizing the missing models problem.

## 3.3 Formalization

We now formalize the missing models problem. We first need to define heuristic weights on each edge $e \in E'$ that give the relative likelihood of the edge $e$ appearing in the model. We want a function $h : \Sigma \to \mathbb{R}_+$, where $h(s)$ for $s \in \Sigma$ represents the heuristic weight of $s$. We then want to extend $h$ to paths $h : \Sigma^* \to \mathbb{R}_+$. For technical reasons we require that if $l = s_1 ... s_n$, then

$$h(l) \geq \max_{1 \leq i \leq n} s_i.$$

Following the notation of [2], we refer to the set of such functions as **superior** functions. For our purposes, it will suffice to define

$$h(s_1 ... s_n) = h(s_1) + ... + h(s_n).$$

We also define $h$ on edges $e \in E \cup E'$ and on paths $p = (e_1, ..., e_n) \in (E \cup E')^*$ by defining $h(e) = h(l_e)$, where $l_e \in \Sigma$ denotes the label on $e$, and $h(p) = h(e_1) + ... + h(e_n)$.

Now that we have assigned heuristic weights to the edges, we can formalize the missing models problem. We want to find the smallest possible set of "possible" edges would induce flow from $v_{\text{source}}$ to $v_{\text{sink}}$. In other words, we want the CFL path $p = (e_1, ..., e_n)$ from $v_{\text{source}}$ to $v_{\text{sink}}$ such that the heuristic weight of the path $h(e_1) + ... + h(e_n)$ is minimized. Formally:

**Problem 3.2.** Given a program $P$ with augmented flows-to graph $G_m$, and vertices $v_{\text{source}}, v_{\text{sink}} \in V$, find

$$\operatorname*{arg\,min}_{p \in \text{paths}(v_{\text{source}}, v_{\text{sink}})} h(p).$$

Returning to our example, we define the function $h$ byWe define $h$ to be

$$h(s) = \begin{cases} 1 \text{ if } s \in \{\text{store}_*, \text{load}_*\} \\ 0 \text{ otherwise} \end{cases}.$$

We refer to this as the **assignment depth heuristic**. Intuitively, this heuristic gives more weight to simpler models. Now it is clear that

$$h\left((\text{source}, x, \text{alias}), (x, z, \text{alias}), (z, \textit{sink}, \text{alias})\right) = 0,$$

and

$$h\left((\text{source}, y, \text{alias}), (x, z, \text{alias}), (z, \textit{sink}, \text{alias})\right) = 0,$$

## 3.4 Reducing CFL Shortest Path to CFL Shortest String

Here, we give reduce Problem 3.2 to a variation of the CFL shortest string problem:

**Problem 3.3.** Consider a context-free language $L$ generated by a grammar with terminals $\Sigma$, nonterminals $\Gamma$, and a weight function $h : \Sigma \to \mathbb{R}_+$, extend $h$ to $\Sigma^*$ by defining

$$h(s_1...s_n) = h(s_1) + ... + h(s_n).$$

The **shortest string problem** for $L$ is to determine the function

$$\mu : \Sigma \cup \Gamma \to \mathbb{R}_+$$

defined by

$$\mu(A) = \min_{l \in L : A \to l} h(l).$$

Here, $A \to l$ means that there is some sequence of productions that generates the variable $A$ from the string $l$.

The general idea is to find, for each $A \in \Sigma \cup V$, the shortest string $l \in L$ such that $A \to l$, where length is weighted by $h$. In order to reduce Problem 3.2 to Problem 3.3, we will construct a new context-free grammar $(\Gamma_G, \Sigma_G, T_G, \Pi_G)$, that incorporates the structure of the graph $G$.

First, we define $\Sigma_G$ and $\Gamma_G$. For each $A \in \Gamma$, we include

$$\{A(u, v)\}_{u,v \in V} \subset \Gamma_G,$$

and for each $s \in \Sigma$, we include

$$\{s(u, v)\}_{(u,v,s) \in E} \subset \Sigma_G.$$

Next, we define $\Pi_G$. We require that the grammar be normalized, i.e. productions take the form $A \to MN$, $A \to M$, or $A \to \epsilon$. Any grammar can be transformed to a normalized grammar in linear time with a linear increase in the number of productions [1]. Once we have a normalized grammar for the language, we include the following productions in $\Pi_G$:

1. for each $A \to \epsilon \in \Pi$, we include

$$\{A(u, u) \to \epsilon\}_{u \in V},$$

2. for each $A \to M \in \Pi$, $M \in \Sigma$ we include

$$\{A(u, v) \to M(u, v)\}_{u,v \in V},$$

   and if $M \in \Sigma$ we include

$$\{A(u, v) \to M(u, v)\}_{(u,v,M) \in E},$$

3. for each $A \to MN \in \Pi$, if $M, N \in \Gamma$ we include

$$\{A(u, w) \to M(u, v)N(v, w)\}_{u,v,w \in V},$$

   if $M \in \Sigma, N \in \Gamma$ we include

$$\{A(u, w) \to M(u, v)N(v, w)\}_{(u,v,M) \in E, w \in V},$$

if $M \in \Gamma, N \in \Sigma$ we include

$$\{A(u,w) \to M(u,v)N(v,w)\}_{u \in V, (v,w,N) \in E},$$

if $M, N \in \Sigma$ we include

$$\{A(u,w) \to M(u,v)N(v,w)\}_{(u,v,M),(v,w,N) \in E},$$

and we include

$$\{T_G \to T(u,v)\}_{u,v \in V}.$$

We refer to the language defined by the grammar $(\Gamma_G, \Sigma_G, T_G, \Pi_G)$ We then have the following lemma:

**Lemma 3.4.** Given a context-free language $L$ and a normalized grammar $(\Gamma, \Sigma, T, \Pi)$ defining it, and a graph $G = (V, E)$ with edges labeled by $\Sigma$, Problem 3.2 for $L$ and $G$ reduces to Problem 3.3 for $L_G$.

## References

[1] David Melski, Thomas Reps, "Incontrovertibility of set constraints and context-free language reachability," *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip.*, pp. 74-89 (1997).

[2] Donald Knuth, "A generalization of Dijkstra's algorithm," *Information Processing Letters* 6(1) pp. 1-5 (1977).