April 2021

EFFICIENT LABELED SEQUENCE GENERATION FROM SYMBOLIC DIRECTED GRAPHS

Lihao Liang

EFFICIENT LABELED SEQUENCE GENERATION FROM SYMBOLIC DIRECTED GRAPHS

ABSTRACT

This publication describes systems and techniques for more efficiently generating, from a binary file such as an executable or shared library, an application-programming-interface-call (API-call) graph, also referred to as a system call (syscall) graph, and to generate n-grams from the API-call graph. Generally, an API-call graph is generated via static analysis of the whole-program control-flow graph of a binary file, and the API-call graph may include symbolic transitions representing internal function calls. Specifically, this publication describes techniques for computing n-grams from an API-call graph that avoids copying of subgraphs of functions represented by symbolic transitions. Avoiding copying of subgraphs of functions represented by symbolic transitions enables faster generation of n-grams with less memory consumption. The generated n-grams can be used in conjunction with machine learning techniques to perform malware detection or other anti-malware techniques.

DESCRIPTION

An API-call graph generated via static analysis of a binary file includes edges that represent API calls. The API calls may include any calls external to the binary as well as well-known statically linked functions. The sequence of edge labels along a given path through an API-call graph is a list of API calls along a potential execution path of the binary. An n-gram is a contiguous sequence of n API calls along a path in the API-call graph. Thus, n-grams of an API-call graph for a binary represent potential behaviors of the binary. As such, the n-grams of a binary can be used in large-scale machine learning to detect the presence of malicious code within the executable or shared library.

An API-call graph may have three types of transitions: API transitions, epsilon transitions, and symbolic transitions. An API transition is an edge labelled with an API call. An epsilon transition is a no-op edge that doesn't represent any function calls and is used exclusively to model control flow. A symbolic transition models an internal function call, which is represented by a subgraph of the API-call graph.

Using a symbolic transition to represent an internal function call replaces the need to make a copy of the subgraph of the internal function call each time the internal function call is encountered in the API-call graph, therefore reducing the amount of memory required to represent and store API-call graphs. However, it may still be necessary to copy subgraphs represented by internal function calls to generate a full API-call graph in order to compute n-grams from the API-call graph.

This publication describes techniques for computing n-grams from an API-call graph containing symbolic transitions that avoids the subgraph copying issue described above by caching n-grams and other auxiliary information about downstream subgraphs for symbolic transitions. Caching n-grams and other auxiliary information obviates the need to recompute n-grams and other auxiliary information when computing n-grams of upstream subgraphs.

The first step for a system to compute n-grams from an API-call graph is to compute the strongly-connected components of the API-call graph, where each strongly-connected component includes mutually recursive functions. The next step is to convert each strongly-connected component into a single graph node, resulting in a directed acyclic graph. Each node in this directed acyclic graph is called a recursive group. The third step is to topologically sort the recursive groups in the directed acyclic graph. These three steps are considered preprocessing steps.

The system may, given the topologically-sorted directed acyclic graph G(V, E) with vertices V and edges E, and a length of n, output the set of n-grams for graph G. The system may iterate through graph G in reverse topological order, starting with the most downstream recursive group of the graph.

As the system iterates through the directed acyclic graph G in reverse topological order, the system computes, for each recursive group in the graph, 1) the set of n-grams from its recursive-group subgraph, which will be part of the final output of the algorithm and 2) for each function F in the recursive-group subgraph, and k = 0, ..., n, the following sets:

- Entry-k-grams of F: the set of k-grams in paths of the recursive-group subgraph that start from the entry node of function F
- Exit-k-grams of F: the set of k-grams in paths of the recursive-group subgraph that end with the exit node of function F
- Function-k-grams of F: the set of k-grams in paths of the recursive-group subgraph that start from the entry node of function F and end with the exit node of function F

For k = 0, the entry-0-gram, exit-0-gram, and function-0-gram may each be an empty 0-gram.

For the most downstream recursive groups, there are no symbolic transitions in their subgraphs. Thus, the set of n-grams of the subgraph for such a recursive group can be computed via other techniques, such as the techniques described in Boulgakov, Alexandre, "Efficient Extraction of N-Grams from a Grammar", Technical Disclosure Commons. Date of Publication: October 29, 2020, and available online at https://www.tdcommons.org/dpubs_series/3721. The same dynamic programming approach can be used to compute entry-k-grams for k = 0, ..., n. Function exit-k-grams may be computed in the same way on an inverted function subgraph.

Function-k-grams are essentially the entry-k-grams at the exit node of the same function subgraph.

For recursive groups other than the most downstream recursive groups, the challenge is to handle symbolic transitions in their subgraphs. To that end, the system may, for a recursive-group subgraph with one or more symbolic transitions:

i. Compute the set of n-grams of the recursive-group subgraph: Let n-gram-V be the set of n-grams that end with node V. Given a symbolic transition from src node to dest node with function F (src ->F dest), store m-grams-src for m = 0, ..., n. Then n-grams-dest is the union of exit-n-grams of function F and the set {m-gram -> function-k-gram | m-gram in m-grams-src of node src, function-k-gram in function-k-grams of F, for all m + k = n }. Finally, add the set { m-gram -> entry-k-gram | m-gram in m-grams-src of node src, entry-k-gram in entry-k-grams of function F, for all m + k = n } to the final result of n-grams.

ii. Compute the entry-k-grams, exit-k-grams, and function-k-grams of function F in the recursive group: Given a symbolic transition from src node to dest node with function F (src ->F dest), let entry-k1-grams-src be the entry-k1-grams that end with node src. Perform a union of entry-k-grams-dest with the set: { entry-k1-gram -> function-k2-gram | entry-k1-gram in entry-k1-grams-src, function-k2-gram in function-k2-grams of function F, for all k1 + k2 = k }. In addition, add the following set { entry-k1-gram -> entry-k2-gram | entry-k1-gram in entry-k1-grams-src, entry-k2-gram in entry-k2-grams of function F, for all k1 + k2 = k } to the final result of entry-k-grams of function F. The exit-k-grams and function-k-grams of function F may be similarly computed.

The system may therefore output the set of n-grams of all recursive groups.

It is noted that the techniques of this disclosure may be combined with any other suitable technique or combination of techniques. As one example, the techniques of this disclosure may be combined with the techniques described in U.S. Patent Application Publication 2009/0249004 A1 by Taifeng Wang, Tie-Yan Liu, Minghao Liu, and Zhi Chen, available online at https://patents.google.com/patent/US20090249004A1. As another example the techniques of this disclosure may be combined with the techniques described in Boulgakov, Alexandre and Ren, Chuangang, "A Machine-Learned Model to Detect Malicious Code Using API-call N-grams From Static Analysis", Technical Disclosure Commons. Date of Publication: September 10, 2020. Available online at https://www.tdcommons.org/dpubs_series/3590. As another example the techniques of this disclosure may be combined with the techniques described in Liu, Xiaojian; Lei, Qian; and Liu, Kehong, "A Graph-Based Feature Generation Approach in Android Malware Detection", Mathematical Problems in Engineering Volume 2020. Date of Publication: May 27, 2020. Available online at https://doi.org/10.1155/2020/3842094. As another example the techniques of this disclosure may be combined with the techniques described in Albarghouthi, Aws, "Software Verification with Program-Graph Interpolation and Abstraction", Graduate Department of Computer Science, University of Toronto. Date of Publication: 2015. Available online at http://pages.cs.wisc.edu/~aws/papers/thesis.pdf. As another example the techniques of this disclosure may be combined with the techniques described in Marceau, Carla, "Characterizing the Behavior of a Program Using Multiple-Length N-grams", NSPW '00: Proceedings of the 2000 Workshop on New Security Paradigms. Date of Publication: 2001. Available online at https://www.nspw.org/papers/2000/nspw2000-marceau.pdf. As another

example the techniques of this disclosure may be combined with the techniques described in Boulgakov, Alexandre, "Static System-Call Graph Generation", Technical Disclosure Commons. Date of Publication: June 21, 2018. Available online at https://www.tdcommons.org/dpubs_series/1272. As another example the techniques of this disclosure may be combined with the techniques described in Boulgakov, Alexandre, "Efficient Extraction of N-Grams from a Grammar", Technical Disclosure Commons. Date of Publication: October 29, 2020. Available online at https://www.tdcommons.org/dpubs_series/3721. As another example the techniques of this disclosure may be combined with the techniques described in Desnos, Anthony; Petrova, Elena; Boulgakov, Alexandre; Neal, Richard; and Mithra, Zubin, "Flow-Graph Analysis of System Calls for Exploit Detection", Technical Disclosure Commons. Date of Publication: June 21, 2018. Available online at https://www.tdcommons.org/dpubs_series/1271.