Technical Disclosure Commons

Defensive Publications Series

September 2020

A Machine-Learned Model To Detect Malicious Code Using API-call N-grams From Static Analysis

Alexandre Boulgakov

Chuangang Ren

A Machine-Learned Model To Detect Malicious Code Using API-call N-grams From Static Analysis

Abstract:

A technique to improve computer security is to test an executable for the presence of malicious code without running the executable. This publication describes systems and techniques for machine learning on application-programming-interface-call (API-call) n-grams from static analysis to automatically determine whether an executable or shared library binary file includes indicators of malicious code. The systems and techniques generate API-call graphs from the file. From the API-call graphs, the systems and techniques generate n-grams. A machine-learned model, using the n-grams, then identifies malicious code or code that performs unwanted behavior.

Keywords:

System call (syscall), syscall graph, API-call graph, control flow graph, graph matching, static analysis, asynchronous entry, non-deterministic finite automaton (NFA), deterministic finite automaton (DFA), regex matching, exploit detection, kernel, vulnerability, malware detection, anti-malware, machine learning, machine-learned model, n-gram, word pair

Background:

A technique, which is referred to as static analysis, to improve computer security is to test an executable or shared library binary file for the presence of malicious code without running the executable. Examples of malicious code include attempts to exploit known vulnerabilities in an operating system. Many techniques generally do not scale well for analyzing a large number of files in a software package (*e.g.*, an operating system). These techniques may also not keep up well with a constantly changing landscape of malicious and unwanted software. Therefore, systems and techniques that can automatically determine whether an executable or shared library binary file includes indicators of malicious code or code that performs unwanted behavior, which is referred to as "potentially harmful code" in this document, is needed.

Description:

This publication describes systems and techniques to implement a machine-learned model on API-call n-grams from static analysis of a binary file for the detection of malicious code or code that performs unwanted behavior. A machine-learned algorithm detects potentially harmful code by generating API-call graphs from binary files of an executable file or a shared library binary file, computing n-grams from the API-call graphs, and using the n-grams to train a machine-learned model to identify and detect potentially harmful code. Figure 1 illustrates this process.

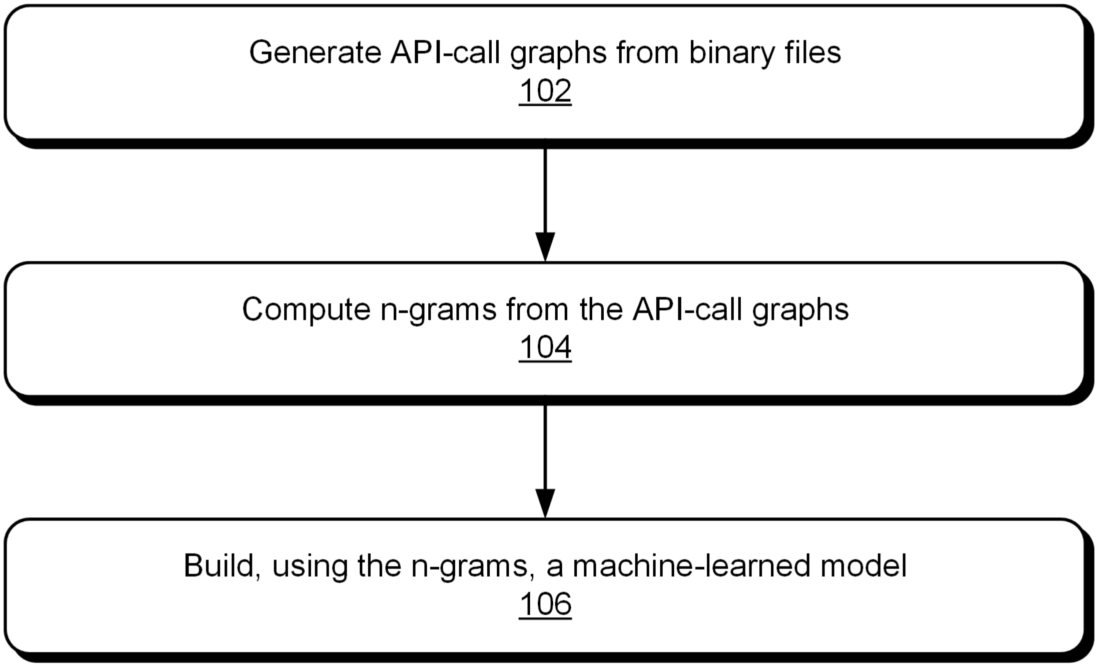
Generate API-call graphs from binary files
102
Compute n-grams from the API-call graphs
104
Build, using the n-grams, a machine-learned model
106

Figure 1

Generate API-call graphs

Figure 2 illustrates a method of generating an API-call graph of a binary file of an executable or shared library given a control-flow graph (CFG) of the executable file.

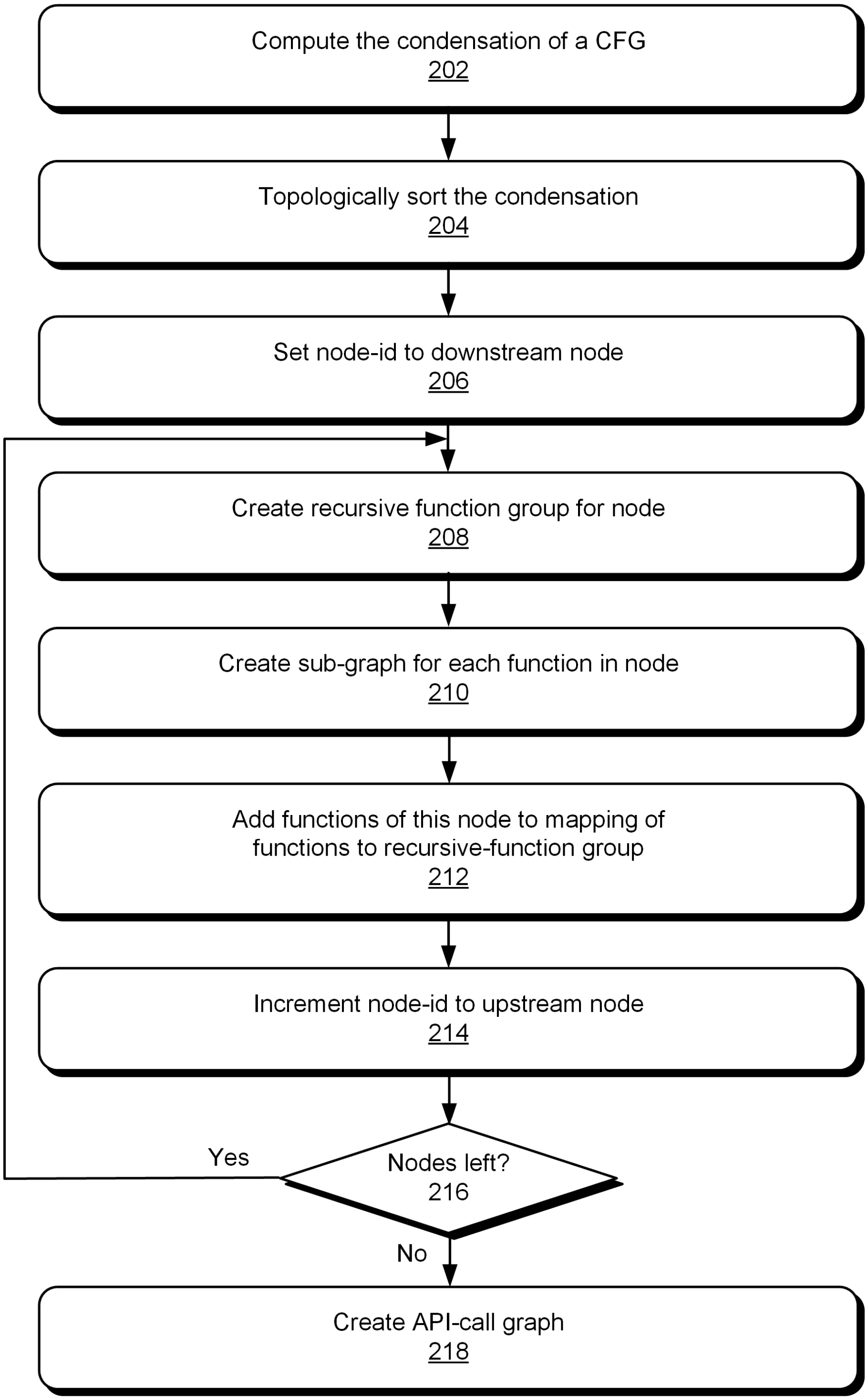
Compute the condensation of a CFG
202
Topologically sort the condensation
204
Set node-id to downstream node
206
Create recursive function group for node
208
Create sub-graph for each function in node
210
Add functions of this node to mapping of functions to recursive-function group
212
Increment node-id to upstream node
214
Yes
Nodes left?
216
No
Create API-call graph
218

Figure 2

The method illustrated in Figure 2 computes the condensation of the CFG (202) to convert mutually-recursive functions into single-graph nodes, resulting in a directed acyclic graph. Each node in the directed acyclic graph has a unique identifier. The condensation is then topologically sorted (204). Within each recursive-function group, the method tracks completed functions. A function is complete when all the functions in the recursive-function group are complete. The topological sort assures that calls to upstream functions (which are incomplete) are not encountered, while downstream functions are complete. Thus, only functions within the current node are potentially incomplete but callable.

The nodes of the condensation are iterated through, starting with the most downstream nodes (206). The method creates a new recursive-function group for a node (208) and populates it with new entry nodes and exit nodes for each function therein. For each function, a subgraph is created (210) for each of its basic blocks, linking them with epsilon transitions based on the control flow. In this context, an epsilon (ε) transition is a no-operations transition, which, in automata theory, is one that occurs without consuming an input symbol). Within a basic block, the control flow is sequential.

Instructions within the basic blocks are translated to transitions as follows:

1. Convert instructions to ε transitions or omit them altogether;
2. Map an assembly-level API-call instruction to the corresponding system call;
3. Map a function call through a procedure-linkage table to an API-call based on the name of the imported function;
4. Map a function-call instruction to the subgraph corresponding to its target as follows:

a. If the function is in the current recursive group, add an ε transition to the entry node and from the exit node.

b. Otherwise, treat the called function as if it were inlined by making a copy of its recursive-function group in the current recursive-function group. The copy is linked-in with an ε transition added to the entry node and from the exit node.

When the functions in the node are processed, the method adds them to the function mapping of recursive-function groups (212). The method then increments the node-id to the immediately-upstream node (214), and, if nodes are left (216), iterates through the remaining nodes.

In this context, an upstream node corresponds to a node that is not downstream. For example, consider a graph with node `a` that has two child-nodes `b` and `c`, resulting in a topological sort `[a, b, c]`. A movement from node `c` to node `b` is considered an upstream movement, even though nodes `b` and `c` are siblings in the graph-theoretic sense.

When the nodes of the condensation are exhausted, an API-call graph is created (218) for the output. The method moves each recursive-function group into the API-call graph by moving the transitions and adding each of the function start nodes to the entry points of the graph. As a result of the condensation, a function can only call other functions that are either within the same node or in downstream nodes. In this way, functions can have their CFGs inlined into upstream callers, which effectively provides return-address tracing and produces a more-accurate CFG. By keeping track of node identifiers within recursive-function groups, the method can efficiently assign new node identifiers with a single update pass over the transitions.

Compute n-grams from the API-call graphs

The systems and techniques then compute n-grams from the generated API-call graph(s). N-grams are used in document processing to summarize the content of a document as a set of text fragments. The generated API-call graphs have document-like content, which is referred to in this document as a grammar. Given a loop in an executable, a grammar can represent a set of execution traces with an arbitrary number of iterations of the loop. The grammar is generally limited to a finite number of n-grams of a given length—at most S^N, where S is the number of symbols in the grammar and N is the length of n-grams. An artificial intelligence (AI) integrator uses the following algorithm to extract n-grams from the API-call graphs contained in the executable file.

The AI integrator uses a recursive algorithm to annotate each node of an API-call graph with its incoming n-grams using dynamic programming and n-graph traversals. The algorithm merges strongly-ε-connected components to make an ε-subgraph acyclic. The nodes of the ε-subgraph are sorted topologically and inverted to obtain a map from a node to its non-ε predecessors (*e.g.*, the source node and API call). For the recursive base case (*e.g.*, if n = 0), the algorithm annotates each node with a set containing the empty 0-gram. Otherwise, the algorithm annotates each node with its incoming (n-1)-grams. The algorithm then annotates each source node and destination node with an array of n-grams, including non-ε predecessors and ε successors. The extracted n-grams are then aggregated per file. An extractor can then output the n-grams directly or as a set of string features in a word form.

Build, using the n-grams, a machine-learned model

String features can be collected and categorized with corresponding potentially harmful code to train a machine-learning model. After sufficient training, the machine-learning model can be deployed to the computer readable medium of a computer system as a machine-learned model. The computer system can use the machine-learned model to recognize potentially-harmful executables. Instead of focusing only on signatures associated with known malicious code, the machine-learned model can target exploits and known vulnerabilities generally to identify potentially-harmful executables.

In this way, the described systems and techniques use machine learning on API-call n-grams to identify malicious and potentially-harmful executables.

References:

[1] Patent Publication: US20130326625A1. Integrating multiple data sources for malware classification. Priority Date: June 5, 2012.

[2] Patent Publication: US20160164901A1. Methods and systems for encoding computer processes for malware detection. Priority Date: December 5, 2014.

[3] Patent Publication: US20190007434A1. Automated detection of malware using trained neural network-based file classifiers and machine learning. Priority Date: June 30, 2017.

[4] Anthony Desnos, Elena Petrova, Alexandre Boulgakov, Richard Neal, and Zubin Mithra. 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.

[5] Alexandre Boulgakov. Static system-call graph generation. Technical Disclosure Commons. Date of Publication: June 21, 2018. Available online at https://www.tdcommons.org/dpubs_series/1272.

[6] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3: A parallel refinement checker for CSP. Int. J. Softw. Tools Technol. Transfer (2016) 18: 149. Available online at https://doi.org/10.1007/s10009-015-0377-y.

[7] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. W. Roscoe. Modelling and analysis of security protocols: the CSP approach. Addison-Wesley (2001).

[8] A. Boulgakov, T. Gibson-Robinson, and A. W. Roscoe. Computing maximal weak and other bisimulations. Formal Aspects of Computing 28(2) pp. 381-407.

[9] Alexandre Boulgakov. Improving Scalability of Exploratory Model Checking. PhD thesis, University of Oxford, 2016. Available online at https://ora.ox.ac.uk/objects/uuid:76acb8bf-52e7-4078-ab4f-65f3ea07ba3d.

[10] A. W. Roscoe. The theory and practice of concurrency. Prentice Hall NJ (1997).

[11] Alexandre Boulgakov. Efficient multi-graph or rooted subgraph matching via merging. Technical Disclosure Commons. Date of Publication: June 18, 2018. Available online at https://www.tdcommons.org/dpubs_series/1252.