June 21, 2018

Static system-call graph generation

Alexandre Boulgakov

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation

Boulgakov, Alexandre, "Static system-call graph generation", Technical Disclosure Commons, (June 21, 2018)
https://www.tdcommons.org/dpubs_series/1272

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
Static system-call graph generation

ABSTRACT

Certain techniques for testing for the presence of malicious code within executables are based on the observation that system calls (syscalls) are a main pathway for exploits. The problem of generating a graph of syscalls of an executable is thus of importance in computer security. The present disclosure describes the generation of the syscall graph of a binary executable given a control flow graph of the executable. Along with a syscall graph, a set of potential asynchronous entry points into the graph is also returned.

KEYWORDS

- system call
- syscall
- syscall graph
- control flow graph
- asynchronous entry
- graph condensation
- anti-malware

BACKGROUND

Certain techniques for testing for the presence of malicious code within executables are based on the observation that system calls (syscalls) are a main pathway for exploits. The problem of generating a graph of syscalls of an executable is thus of importance in computer security.
DESCRIPTION

Given a control flow graph (CFG) of a binary executable file, the techniques of this disclosure efficiently derive a syscall graph together with a set of potential asynchronous entry points into the graph.

![Flowchart](image.png)

Fig. 1: Generating a syscall graph given a CFG
Fig. 1 illustrates generating a system call graph of a binary executable given the control flow graph of the executable. The condensation of the CFG is computed (102) in order to convert all mutually recursive functions into single graph nodes, resulting in a directed acyclic graph. Each node in this directed acyclic graph has a unique identifier. The condensation is topologically sorted (104). Within each inlineable recursive function group, completed functions are tracked. Conversely, for each completed function, the recursive function group that includes it is tracked. A function is only complete when all the functions in the mutually recursive function group are complete. A recursive function group tracks the identifiers of the nodes it contains. 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 (106). A new inlineable recursive function group is created for a node (108), and is populated with fresh entry and exit nodes for each function it contains. For each function in a node, a subgraph is created (110) for each of its basic blocks, linking them with epsilon transitions according to the control flow. In this context, an epsilon transition is a no-operations transition, e.g., in the sense of automata theory, one that occurs without consuming an input symbol. Within a basic block, control flow is sequential except in the case of function calls. Instructions within basic blocks are translated to transitions as follows:

1. Most instructions are converted to epsilon transitions or omitted altogether.
2. An assembly-level syscall instruction is mapped to the corresponding system call.
3. A function call through the procedure linkage table is mapped to a system call based on the name of the imported function.

4. A function call instruction is mapped to the subgraph corresponding to its target as follows.
   a. If the function is in the current recursive group an epsilon transition is added to the function's entry node and from its exit node.
   b. Otherwise the called function is treated 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 epsilon transition added to its entry node and from its exit node.

When all the functions in this node have been processed, they are added to the mapping of functions to recursive function groups (112). The node-id is incremented to the immediate upstream node (114), and if nodes are left (116), iteration through the condensation continues.

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, a syscall graph is created (118) for the output, as follows. Each recursive function group is moved into it by moving its transitions and adding each of its function start nodes to the graph's entry points. As a result of the condensation, a function can only call other functions that are either within the same node or in more downstream nodes. This means that functions can have their CFGs inlined into upstream
callers, which effectively provides return address tracing and produces a more accurate CFG. Keeping track of node identifiers within recursive function groups allows new node identifiers to be assigned efficiently with a single update pass over the transitions. The techniques are optimized by storing each recursive function group's transitions into a graph as they are added, by additionally recording the first and last nodes in the subgraph, and the parent graph, within each recursive function group.

The techniques of this disclosure can be applied to analyze executable or shared library files to determine if such executables include malicious code that attempts to exploit known vulnerabilities. Such data can be used in making security-related decisions about the executables. Online application stores or other platforms that offer third-party applications or software can use the techniques to test software that is submitted for download or sale to end-users. The techniques also find applicability in the computer security industry, e.g., within anti-virus, anti-malware, and other security products.

CONCLUSION

Certain techniques for testing for the presence of malicious code within executables are based on the observation that system calls (syscalls) are a main pathway for exploits. The problem of generating a graph of syscalls of an executable is thus of importance in computer security. The present disclosure describes the generation of the syscall graph of a binary executable given a control flow graph of the executable. Along with a syscall graph, a set of potential asynchronous entry points into the graph is also returned.
REFERENCES


