Gray-Box Extraction of Execution Graphs for Anomaly Detection

Debin Gao*
dgao@ece.cmu.edu

Michael K. Reiter†
reiter@cmu.edu

Dawn Song†
dawnsong@cmu.edu

ABSTRACT

Many host-based anomaly detection systems monitor a process by observing the system calls it makes, and comparing these calls to a model of behavior for the program that the process should be executing. In this paper we introduce a new model of system call behavior, called an *execution graph*. The execution graph is the first such model that both requires no static analysis of the program source or binary, and conforms to the control flow graph of the program. When used as the model in an anomaly detection system monitoring system calls, it offers two strong properties: (i) it accepts only system call sequences that are consistent with the control flow graph of the program; (ii) it is maximal given a set of training data, meaning that any extensions to the execution graph could permit some intrusions to go undetected. In this paper, we formalize and prove these claims. We additionally evaluate the performance of our anomaly detection technique.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems, Security and Protection

General Terms

Security

Keywords

Intrusion detection, Anomaly detection, System call monitoring, Control flow graph

*Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
†Department of Electrical & Computer Engineering, Department of Computer Science and CyLab, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
This work was supported by the U.S. National Science Foundation.

1. INTRODUCTION

Many host-based intrusion detection systems (e.g., [4, 11, 17, 18]) and related sandboxing and confinement systems (e.g., [12, 19]) monitor the system calls emitted by a process in order to detect deviations from a previously constructed model of system call behavior. We coarsely divide these systems into "black box", "gray box" and "white box" approaches, based on the information they use to build the model to which they compare system calls at run time. On the one hand, black-box and gray-box methods build a model of system-call behavior by monitoring sample executions. Within this space, black-box detectors employ only the system call number (and potentially the arguments, though we do not consider arguments in this paper) that pass through the system call interface when system calls are made (e.g., [4, 16]). A gray-box detector extracts additional runtime information from the process, e.g., by looking into the process' memory (e.g., [3, 15]). On the other hand, white-box approaches obtain the model by statically analyzing the source code or binary (e.g., [2, 6, 7, 18]).

By their nature, black-box and gray-box detectors detect *anomalous* behavior, i.e., behavior different from "normal" runs, regardless of whether it results from an intrusion or an execution path that was not encountered during training. In contrast, white box detectors detect actual deviations from the program text, for which an intrusion is virtually the only conceivable explanation (assuming that the program is not self-modifying). As such, white-box detectors can be designed to have a zero false-positive rate, in the sense that an alarm always indicates an intrusion. Since minimizing false positives is a significant factor in gaining user acceptance, this is an important advantage of white-box approaches.

White-box approaches, however, are not always viable. First, source code is often not available, and the complexity of performing static analysis on, e.g., x86 binaries is well documented.[1] In fact, all white-box intrusion detection sys-

[1]This complexity stems from difficulties in code discovery and module discovery [13], with numerous contributing factors, including: variable instruction size (Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [10]); hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [14]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [10, 13]. Due to these issues and others, binary analysis/rewrite tools for the x86 platform have strict restrictions on their applicable targets [9, 10, 13, 14].

tems of which we are aware (including those cited above) have eschewed this common platform. Static analysis is also difficult for programs protected by obfuscation or digital rights management (DRM) technologies that are designed to render static analysis of control flow all but impossible (e.g., [1]). Finally, white-box techniques can fail on self-modifying programs. We thus believe that examination of gray-box and black-box approaches can play an important role where white-box approaches are unavailable.

In this paper we present a new gray-box model, called an *execution graph*, that is, to our knowledge, the first gray- or black-box technique for which a positive relationship to what is achievable via common white-box techniques can be proved analytically. Intuitively, the goal that we set for our technique is to build a model that accepts the same system call sequences as would be accepted by a model built from the control flow graph of the program, which is the basis of many white-box techniques. This, of course, is not achievable, since our gray-box technique can train only on observed runs of the program, which may miss entire branches of the program that static analysis would uncover. Nevertheless, using gray-box techniques alone, our approach constructs an execution graph with the following two useful properties: First, the system call sequences (the language) accepted by the execution graph are a subset of those accepted by the control flow graph of the program. Second, the language accepted by the execution graph is *maximal* for the training sequences it was provided. Specifically, we show that there exists a program of which the control flow graph would accept the same language as the execution graph. In other words, if the execution graph were to accept any other system call sequence s, then there is a program that can emit exactly the same training sequences but for which the control flow graph would not accept s.

In some sense, this is the best one can hope to achieve toward using a gray-box technique to mimic the power of a control flow model obtained via white-box analysis. Moreover, as the control flow model that we set for our goal is equivalent to the most restrictive white-box models known in the literature—the model is sensitive not only to the sequence of system calls, but the sequence of active function calls when each system call occurs—our approach mimics some of the best white-box techniques known today, using only gray-box analysis. Additionally, we demonstrate through a prototype implementation that monitoring via an execution graph is very efficient.

The rest of the paper is organized as follows. Section 2 discusses related work in this area. Section 3 defines an execution graph, and the language accepted by an execution graph. We show the claimed relationships between execution graphs and control flow graphs in Section 4. Performance evaluations are discussed in Section 5. Finally, we conclude in Section 6.

2. RELATED WORK

Numerous white-box approaches to intrusion detection have focused on monitoring a process' system-call conformance with the control flow graph of the program it is ostensibly running. One of the earliest works, due to Wagner and Dean [18], generates a range of models based on the control flow graph of the program, generated via static analysis of the source. Their most accurate model, equivalent to the control flow model that we adopt here (Section 4), resulted in very substantial runtime monitoring overheads. This cost, as well as the need for analyzing source code, were addressed in following works due to Giffin et al. [6, 7] and Feng et al. [2]. These works included modifying the binary to permit the runtime monitor to perform more efficiently.

Black-box approaches were pioneered by Forrest et al. [4], who introduced an approach to characterize normal program behavior in terms of sequences of system calls. System call sequences are broken into patterns of fixed length, which are learned and stored in a table. Wespi et al. [20, 21] extend this approach to permit variable-length patterns of system calls. To our knowledge, Sekar et al. [15] proposed the first gray-box approach, coupling the system call number with the program counter of the process when the system call is made. Feng et al. [3] proposed extending the gray-box information used to include return addresses on the call stack of the process when a system call is made. While the benefits and costs of many of these approaches have been studied [5], the behavior of none of these approaches has been formally related to that of the white-box system call models. In fact it is generally easy to confirm that these prior black- and gray-box models neither contain nor are contained by the white-box models, in terms of the languages of system call sequences they accept. Black-box approaches have also been extended to monitor system call arguments [8], however we do not consider them in the paper.

3. EXECUTION GRAPHS

In this section we describe our model, called an *execution graph*, for anomaly detection, which is built using a gray-box technique. Our technique assumes that the program being monitored is implemented in a programming language for which the runtime utilizes a call stack, where each stack frame corresponds to a function call in the program and includes a return address. Every implementation of the C and C++ programming languages known to us satisfies this criteria, and these languages are the primary motivations for our work.

The execution graph technique we describe in this paper works, during both training and monitoring, by observing system calls along with additional runtime information that it extracts upon each system call, namely the return addresses on the call stack of the monitored process when the system call is made. We define a system call along with the return addresses on the call stack when a system call is made as an *observation*. Each such observation can be represented by an arbitrary-length vector of integers, each in the range of $[0, 2^{32})$ assuming a 32-bit platform. The last element of the vector is the system call number, and the preceding elements are the return addresses on the call stack when the system call is made, with the first address being an address in `main()`, i.e., an address in the first function executed.

We formally define the concept of observation below, and we call a sequence of observations an *execution*.

Definition 3.1 (Observation, execution). *An* observation *is a tuple of positive integers $\langle r_1, r_2, \ldots, r_k\rangle$, where $k > 1$. An* execution *is an arbitrary-length sequence of observations.* □

In particular, for an observation $\langle r_1, r_2, \ldots, r_k\rangle$, r_1 is an address in `main()`, r_{k-1} is the "return address" which corresponds to the instruction that makes the system call,[2] and r_k is the system call number.

We next introduce the concept of an execution graph, which is built by observing executions as defined above. The goal we set for this new model, as briefly stated in Section 1, is to build a model that accepts the same system call sequences that will be accepted by most models built from white-box techniques. Informally, we need to extract function call structures from observations so that a graph similar to the control flow graph can be built. To achieve this we analyze every two consecutive system calls and the return addresses on the call stack when each system call is made, i.e., to analyze two consecutive observations. Since each return address represents a stack frame, consecutive observations reveal some information about the function call structure of the program. In the following definition, we show how this information is used to build an execution graph.

The execution graph is one of the most important concepts in this paper, especially the inductive definition of the edges in the graph. Intuitively, we use E_{rtn} to represent the returning of a function to its calling location, use E_{crs} to represent the execution flow within a function, and use E_{call} to represent the calling from a function call site to its call target. These three sets of edges are defined in the base case by processing consecutive observations. The inductive part of the definition is used to post-process these sets of edges, and to discover "missing" edges, where the relationship between two nodes could be derived from the executions, but not by processing any individual pair of observations. (This induction is further explained in an example after we formally present the definition.)

Definition 3.2 (Execution graph, leaf node, $\overset{\mathsf{crs}}{\to}$). *An* execution graph *for a set of executions $\mathcal{X}$ is a graph* EG$(\mathcal{X}) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$*, where V is a set of nodes, and E_{call}, E_{crs}, $E_{\mathsf{rtn}} \subseteq V \times V$ are directed edge sets, defined as follows:*

- *For each execution $X \in \mathcal{X}$ and each observation $\langle r_1, r_2, \ldots, r_k\rangle \in X$, V contains nodes $r_1, r_2, \ldots, r_k$. r_k is called a* leaf node *of the* execution graph EG$(\mathcal{X})$*. In the case where $\langle r_1, r_2, \ldots, r_k\rangle$ is the first observation in an execution, r_k is also denoted as an* enter *node; in the case where $\langle r_1, r_2, \ldots, r_k\rangle$ is the last observation in an execution, r_k is also denoted as an* exit *node. (Note that an execution graph could have more than one* enter *node and more than one* exit *node.)*

- *The sets E_{call}, E_{crs}, E_{rtn} are defined inductively to contain only edges obtained by the following rules:*

  – **(Base case)** *For each execution X in $\mathcal{X}$ and each pair of consecutive observations $\langle r_1, r_2, \ldots, r_k\rangle$, $\langle r'_1, r'_2, \ldots, r'_{k'}\rangle$ in X,*

$$\begin{aligned} E_{\mathsf{rtn}} &\leftarrow E_{\mathsf{rtn}} \cup \{(r_{i+1}, r_i)\}_{\ell \le i < k} \\ E_{\mathsf{crs}} &\leftarrow E_{\mathsf{crs}} \cup \{(r_\ell, r'_\ell)\} \\ E_{\mathsf{call}} &\leftarrow E_{\mathsf{call}} \cup \{(r'_i, r'_{i+1})\}_{\ell \le i < k'} \end{aligned}$$

  where

$$\ell = \begin{cases} k-1 & \text{if } \langle r_1, r_2, \ldots, r_k\rangle = \langle r'_1, r'_2, \ldots, r'_{k'}\rangle \\ \left(\arg\max_j : \langle r_1, r_2, \ldots, r_j\rangle = \langle r'_1, r'_2, \ldots, r'_j\rangle\right) + 1 & \text{otherwise} \end{cases}$$

  If r_k is an enter *node,*

$$E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(r_i, r_{i+1})\}_{1 \le i < k}$$

  If $r'_{k'}$ is an exit *node,*

$$E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(r'_{i+1}, r'_i)\}_{1 \le i < k'}$$

  – **(Induction)** *Define the relation $r \overset{\mathsf{crs}}{\to} r'$ to be true if there exists a path from r to r' consisting of only edges in E_{crs}.*

    ∗ *If $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \overset{\mathsf{crs}}{\to} x_2$, and $(x_2, x_3) \in E_{\mathsf{rtn}}$, then $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_2, x_0)\}$ and $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_3, x_1)\}$;*

    ∗ *If $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \overset{\mathsf{crs}}{\to} x_2$, and $(x_3, x_2) \in E_{\mathsf{call}}$, then $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_3, x_1)\}$ and $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_0, x_2)\}$;*

    ∗ *If $(x_1, x_0) \in E_{\mathsf{rtn}}$, $x_1 \overset{\mathsf{crs}}{\to} x_2$, and $(x_2, x_3) \in E_{\mathsf{rtn}}$, then $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_1, x_3)\}$ and $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_2, x_0)\}$.*

□

Note that the integers in an observation serve as labels for the nodes created. For simplicity, we do not differentiate a node and its label, i.e., in the above definition, r and x denote both the nodes and their labels.

Example 3.1 illustrates the reason why such an inductive definition is necessary. (Source code and the execution graph of Example 3.1 are shown in Figure 1.)

Example 3.1. *In this example,* `f()` *is called twice from* `main()`*, while each time it is called not all instructions in* `f()` *are executed. Some execution paths of the program might not be uncovered, e.g., due to the fixed values of* `a` *and* `b` *in the executions. In this example, edges* `(f.3, main.5)` *and* `(f.5, main.3)` *can only be obtained by the inductive definition in Definition 3.2.*

With the inductive definition in Definition 3.2, execution graph as shown in Figure 1 can be obtained even if the value of `a` *and* `b` *are fixed in executions $\mathcal{X}$.*[3] □

Definition 3.3 ($\overset{\mathsf{call}}{\to}$, $\overset{\mathsf{rtn}}{\to}$). *Let* EG$(\mathcal{X}) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *be an execution graph. $r \overset{\mathsf{call}}{\to} r'$ iff there exists a path from r to r' consisting of only edges in E_{call}. $r \overset{\mathsf{rtn}}{\to} r'$ iff there exists a path from r to r' consisting of only edges in E_{rtn}.* □

Recall that we want to mimic the power of the most restrictive control flow graph model known in the literature, where not only the system call sequence, but also the sequence of active function calls when each system call occurs, are captured. To do this, we use the notion of *execution stack* to capture the active function calls allowed in an execution graph.

[2]Though on most platforms, system calls are implemented differently from function calls, r_{k-1} can be retrieved from the stack in a similar fashion, and we still refer to it as a return address.

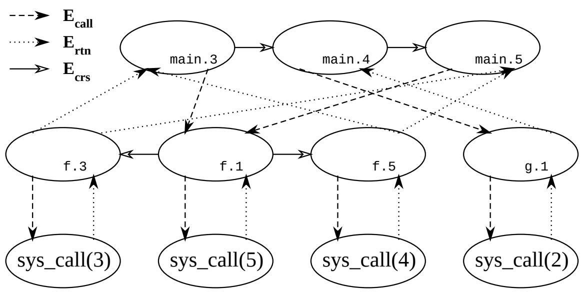
[3]Nodes in an execution graph are typically denoted by integers only. In Figure 1 we show the correspondence with the line number and function name, for the purpose of illustration.

```
   int main(int argc, char *argv[]) {
1:    int a, b;
2:    a = 1; b = 2;
3:    f(a);
4:    g();
5:    f(b);
   }

   void f(int x) {
1:    sys_call(5);
2:    if (x == 1)
3:       sys_call(3);
4:    else if (x == 2)
5:       sys_call(4);
   }

   void g() {
1:    sys_call(2);
   }
```

(a) source code of Example 3.1



(b) execution graph of Example 3.1

Figure 1: Source code and execution graph of Example 3.1

Definition 3.4 ($\overset{\text{xcall}}{\to}$, Execution stack). *Let* EG($\mathcal{X}$) = (V, E_{call}, E_{crs}, E_{rtn}) *be an execution graph.* $r \overset{\text{xcall}}{\to} r'$ *iff*

- $(r, r') \in E_{\text{call}}$*; or*
- *There exists a node* $r'' \in V$*, such that* $(r, r'') \in E_{\text{call}} \wedge r'' \overset{\text{crs}}{\to} r'$.

An execution stack *in* EG($\mathcal{X}$) *is a sequence of nodes* $\langle r_1, r_2, \ldots, r_n\rangle$*, such that*

- *For each* $1 \le i < n, r_i \overset{\text{xcall}}{\to} r_{i+1}$*; and*
- r_1 *corresponds to an address in* `main()`*, i.e., an address in the first function executed; and*
- r_n *is a leaf node.*

□

Intuitively, an execution stack captures a system call and the active functions (functions that have not returned) when the system call is made, which is also what an observation captures. However, an execution stack might or might not have a corresponding observation in the executions $\mathcal{X}$ that are used to construct the execution graph EG($\mathcal{X}$).

We next define the notion of *successor*. Intuitively, if observation x' follows another observation x in an execution, then x' corresponds to an execution stack that is a *successor* of the execution stack corresponding to x. The notion of *successor* in an execution graph defines whether a system call (and the corresponding active functions) are allowed to follow another system call.

Definition 3.5 (Successor). *Execution stack* $s' = \langle r'_1, \ldots, r'_{n'}\rangle$ *is a* successor *of execution stack* $s = \langle r_1, \ldots, r_n\rangle$ *in an execution graph* EG($\mathcal{X}$) = (V, E_{call}, E_{crs}, E_{rtn}) *if there exists an integer k such that* $r_n \overset{\text{rtn}}{\to} r_k$, $(r_k, r'_k) \in E_{\text{crs}}$, $r'_k \overset{\text{call}}{\to} r'_{n'}$, *and for each* $1 \le i < k$, $r_i = r'_i$. □

Definition 3.6 (Execution path). *An* execution path δ *is a sequence of execution stacks in an execution graph* EG($\mathcal{X}$) = (V, E_{call}, E_{crs}, E_{rtn})*, say* $\delta = \langle s_1, s_2, \ldots, s_n\rangle$, $s_i = \langle r_{i,1}, r_{i,2}, \ldots, r_{i,m_i}\rangle$*, where*

- *For each* $1 \le i < m_1$, $(r_{1,i}, r_{1,i+1}) \in E_{\text{call}}$*; and*
- r_{1,m_1} *is an* enter *node; and*
- *For each* $1 \le i < n$, s_{i+1} *is a successor of* s_i.

□

Intuitively, an execution path is a sequence of execution stacks that corresponds to a possible execution of the program that emitted the executions $\mathcal{X}$. Notice that it only requires the sequence of execution stacks to be allowed by the execution graph (captured by the notion of successor, which is defined in Definition 3.5), which might or might not have appeared in the executions $\mathcal{X}$ from which the execution graph EG($\mathcal{X}$) is built.

Definition 3.7 (Language accepted by EG($\mathcal{X}$)). *The* language accepted by EG($\mathcal{X}$)*, denoted* $L_{\text{EG}(\mathcal{X})}$*, is the set of all execution paths in* EG($\mathcal{X}$). □

Each string in the language accepted by an execution graph is a sequence of execution stacks. Each execution stack consists of a sequence of integers, which intuitively represents a system call and the return addresses of the active functions when the system call is made. Though we have defined execution graphs built from observations including return addresses, they also have a black-box variant: In the case where only the system call number is used to describe a system call (return addresses are not extracted), an execution stack consists of only one integer, which is the system call number. Consequently a string in the language will become a sequence of system call numbers. We do not discuss this variation further.

4. PROPERTIES OF EXECUTION GRAPHS

We briefly stated in Section 1 that the goal of our technique is to build a model that accepts system call sequences that would be accepted by a model built from the control flow graph of the program. In this section, we formalize two important properties of an execution graph. First, it accepts only system call sequences that are consistent with the control flow graph of the program. Second, it is maximal given a set of training data, meaning that any extensions to an execution graph could permit some intrusions to go undetected. To formalize these two properties, we first define control flow graphs, the language a control flow graph accepts, and well-behaved executions. With these definitions, the theorems are presented in Section 4.3.

4.1 Control Flow Graphs

A control flow graph is an abstract representation of a procedure or program. In this paper, it is convenient to consider a variation on the traditional control flow graph for a program P, denoted CFG(P). First, CFG(P) consists of a number of *control flow subgraphs*, one per function F in P, denoted CFSG(F). Second, since we are interested only in function calls and system calls in P, each CFSG(F) has one node per function call and two nodes per system call that it contains, in addition to its entry and exit node, and no other nodes. Though these variations render CFG(P) different from a traditional control flow graph, we will still refer to it as one.

In this paper, we refer to a *jump* as a nonsequential transfer of control, distinct from a function call or a system call. With this, we define the relationship between two instructions in a function.

DEFINITION 4.1 (FOLLOW). *Instruction t'* follows *instruction t iff t and t' are in the same function and*

- **(Base case)** *t' is at a higher address than t, and there is no jump, function call or system call between t and t';*
- **(Induction)** *There exists a jump c and a corresponding jump target c', such that t'* follows *c' and c* follows *t.*

□

The above definition defines the relative position of two instructions in a function. Next we define control flow subgraph (CFSG) and call nodes in a CFSG. In order to simplify the definition, we assuming that there are two no-op instructions in each function F denoting the starting and ending of F respectively.

DEFINITION 4.2 (CONTROL FLOW SUBGRAPH, CALL NODE). *A* control flow subgraph *for a function F is a directed graph* CFSG(F) $= (V, E)$. *V contains*

- *A* function call node *per function call in F;*
- *A* system call node *per system call in F;*
- *A* system call number node *per system call in F;*
- *A designated F.*enter *node and a designated F.*exit *node.*

Function call nodes *and* system call nodes *are the* call nodes *of* CFSG(F). *Each node is identified by a label.* $(u, v) \in E$ *iff*

- *The instruction that corresponds to v follows (as defined in Definition 4.1) the instruction that corresponds to u; or*
- *u is a* system call node *and v is the corresponding* system call number node.

□

Each node in a CFSG has a label. The label of a call node could be assigned as the address of the instruction that immediately follows the call if static analysis is applied on binaries, as assumed in Section 4.3 for convenience. The label of a system call number node is the corresponding system call number. As in the definition of execution graphs, we do not differentiate a node and its label, i.e., in the above definition, u and v denote both the nodes and their labels.

The control flow graph CFG(P) of a program P is obtained by connecting control flow subgraphs of each function in P together to form a new graph.

DEFINITION 4.3 (CONTROL FLOW GRAPH). *Let P be a program consisting of functions $F_1, F_2, \ldots, F_n$. Let* CFSG(F_i) $= (V_i, E_i)$ *denote the control flow subgraph for F_i. The* control flow graph *for P is a directed graph* CFG(P) $= (V, E)$, *where $V = \bigcup_i V_i$ and where $(u, v) \in E$ iff*

- *For some $1 \leq i \leq n$, $(u, v) \in E_i$; or*
- *$v = F_i$.*enter *and u is a function call node representing a call to F_i; or*
- *$u = F_i$.*exit *and v is is a function call node representing a call to F_i.* □

Figure 2 shows the control flow graph of the program in Example 3.1 (the source code is shown in Figure 1).

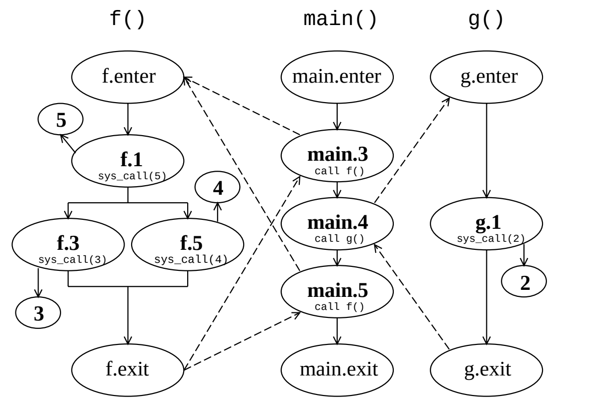


Figure 2: Control flow graph of Example 3.1

A control flow graph as described above defines all possible executions of a program P, in terms of the function and system calls it makes. During program execution, nodes in the control flow graph are traversed by following the directed edges. An execution of the program can be described by a path through which the nodes are traversed. A *call cycle* corresponds to the calling and returning of a function.

DEFINITION 4.4 (CALL CYCLE). *A sequence of nodes $\langle v_1, v_2, \ldots, v_n \rangle$ in* CFG(P) $= (V, E)$ *is a* call cycle *iff for some function $F \in P$ and the corresponding* CFSG(F) $= (V_F, E_F)$ *in* CFG(P),

- **(Base case)**
  - *$v_1 = v_n$ is a function call node representing a call to F, $v_2 = F$.enter, $v_{n-1} = F$.exit; and*
  - *For each $1 < i < n-1$, $v_i \in V_F$; and*
  - *For each $1 \leq i < n$, $(v_i, v_{i+1}) \in E$.*
- **(Induction)** *For some integers k and k', where $1 < k < k' < n-1$,*
  - *$v_1 = v_n$ is a function call node representing a call to F, $v_2 = F$.enter, $v_{n-1} = F$.exit; and*
  - *For each $1 \leq i < n$, $(v_i, v_{i+1}) \in E$; and*
  - *For each $1 < i \leq k$ and $k' \leq i < n-1$, $v_i \in V_F$; and*
  - *$\langle v_k, v_{k+1}, \ldots, v_{k'} \rangle$ is a call cycle.*

□

A *series of call cycles* is a concatenation of at least one call cycle.

Definition 4.5 (Observable path). *An* observable path *π in* CFG*$(P) = (V, E)$ is a sequence of nodes, say $\langle v_1, v_2, \ldots, v_n \rangle$, where*

- *$v_1 =$ main.enter, i.e., the entry node for the first function called in the program; and*
- *v_n is a system call node; and*
- *For each $1 \leq i < n$, $(v_i, v_{i+1}) \in E$; and*
- *For each $1 < i < n$, if v_i is a function call node representing a call to F, then either $v_{i+1} = F$.enter or $v_{i-1} = F$.exit; if $v_{i-1} = F$.exit, then there exists an integer i', where $1 < i' < i$, such that $\langle v_{i'}, v_{i'+1}, \ldots, v_i \rangle$ is a call cycle.*

□

The path defined in Definition 4.5 is called *observable* because it induces a system call, and thus intuitively would be visible to an intrusion detection system monitoring system calls. Numerous white-box process monitors additionally keep track of the active function calls in the process running the program, based on information gathered from static analysis of the program. We define active calls on an observable path as follows.

Definition 4.6 (Active calls on an observable path). *Let $\pi = \langle v_1, v_2, \ldots, v_n \rangle$ be an observable path in* CFG*$(P) = (V, E)$. We define the sequence of* active calls *on π, denoted $A(\pi)$, to be the result of the following procedure.*

1. *Delete all call cycles on π;*
2. *Denote the remaining nodes by $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_k} \rangle$, where for each $1 \leq j < k$, $i_j < i_{j+1}$. For each $1 \leq j < k$, delete v_{i_j} unless v_{i_j} is a function call node;*
3. *Append a node v_{n+1} to the end of the sequence, where v_{n+1} is the system call number node such that $(v_n, v_{n+1}) \in E$.*

□

Since v_n (the last node on an observable path) does not belong to any call cycles, it is not deleted in the first step of the procedure in Definition 4.6. As such, $v_{i_k} = v_n$ in the second step of the procedure in Definition 4.6, and this node is not deleted in the second step either (since only nodes v_{i_j} for $1 \leq j < k$ are eligible to be deleted). In other words, v_n is always the second last element in the output of $A(\pi)$, with the last element being the system call number.

Definition 4.7 (Language accepted by CFG(P)). *Let Π be the set of all observable paths in* CFG*(P), and for any $\pi \in \Pi$, let $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$ denote all the observable prefixes of π in order of increasing length, where $\pi_n = \pi$. Then, the* language accepted by CFG(P) *is*

$$L_{\mathrm{CFG}(P)} = \{\langle A(\pi_1), \ldots, A(\pi_n) \rangle : [\exists \pi \in \Pi : \mathsf{pre}(\pi) = \langle \pi_1, \ldots, \pi_n \rangle]\}$$

□

Notice that we define the language accepted by CFG(P) in terms of the system calls it makes and the active functions when each system call is made. A string in the language is a sequence of symbols, each of which describes a system call made by the program.

Example 4.1. *Figure 3 shows the source code and the control flow graph of a very simple program, which consists of four functions and makes four different system calls. In the program shown in Figure 3, the second system call made is* `read`*, which corresponds to the system call number* 3*. The following is an observable path from* main.enter *to the node that makes this system call.*

$$\pi_1 = \langle \texttt{main.enter}, \texttt{main.1}, \texttt{main.2}, \texttt{f.enter}, \texttt{f.1}, \texttt{g.enter}, \ldots, \texttt{g.exit}, \texttt{f.1}, \texttt{f.2} \rangle$$

When trying to find the active calls on π_1, f.1, g.enter, …, g.exit, f.1 *should be deleted in the first step of the procedure in Definition 4.6, since they correspond to a call cycle (a completed function call).* main.enter *and* main.1 *should be deleted in the second step of the procedure in Definition 4.6, as they are not function call nodes. Therefore,*

$$A(\pi_1) = \langle \texttt{main.2}, \texttt{f.2}, 3 \rangle$$

In this example, the language accepted by the control flow graph is

$$L_{\mathrm{CFG}(Ex[4.1])} = \{\langle \texttt{main.1}, 5 \rangle, \langle \texttt{main.2}, \texttt{f.2}, 3 \rangle, \langle \texttt{main.2}, \texttt{f.3}, \texttt{h.1}, 4 \rangle, \langle \texttt{main.2}, \texttt{f.3}, \texttt{h.2}, 6 \rangle\}$$

Figure 4 shows an execution graph built from executions of the program in Example 4.1, assuming the input covers all possible paths of the program.[4]

□

Notice that the languages accepted by the execution graphs of the two examples (Example 3.1 and Example 4.1) are very similar to the languages accepted by their control flow graphs. (In particular the only differences are the values of

[4]Nodes in an execution graph are denoted by integers only. In Figure 4 we show the correspondence with nodes in the control flow graph of the program, i.e., the line number and function name, for the purpose of illustration.

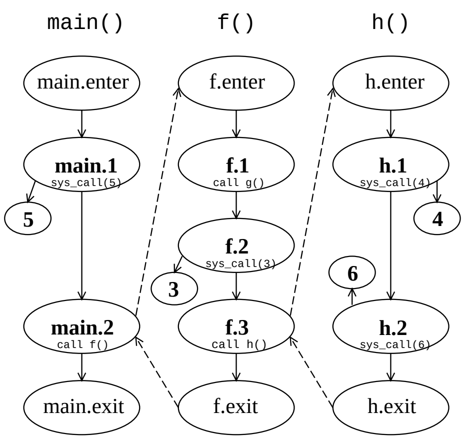
```
   int main(int argc, char *argv[]) {
1:    sys_call(5);
2:    f();
   }

   void f() {
1:    g();
2:    sys_call(3);
3:    h();
   }

   void h() {
1:    sys_call(4);
2:    sys_call(6);
   }

   void g() {
      ...
   }
```

(a) source code of Example 4.1



(b) control flow graph of Example 4.1 (CFSG(`g()`) is not shown)

Figure 3: Source code and the control flow graphs of Example 4.1

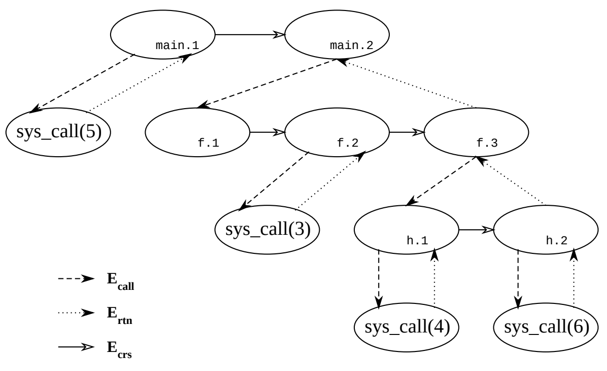


Figure 4: Execution graph of Example 4.1

the labels.) Intuitively this similarity is what we are trying to achieve and why execution graphs are very useful in anomaly detection. In Section 4.3 we will formally introduce the relationship between the two, by showing two very useful properties of execution graphs.

4.2 Well-Behaved Executions

To this point in the paper, we have not specified the program executions that are *useful* to build an execution graph (though any execution results in one). However, to prove a relationship with the control flow graph of the program, it is necessary to specify which executions are useful for this purpose. Intuitively, these executions are ones that do not include an attack, and more specifically, for which the return addresses are a reliable reflection of the intended execution of the underlying program. We refer to such executions as *well-behaved*.

More precisely, denote the execution of program P on input I by $P(I)$. Input string I includes all inputs to the process running P since its initialization, and can include multiple "invocations" if program P is a server program. In this case, the multiple invocations of P are separated in I in a canonical way. The runtime process that executes $P(I)$ maintains a call stack in conformance with certain conventions, induced via the function call and return code emitted by the compiler for the language. While we do not detail these conventions here, we expect that the return address of each stack frame is inserted when the function call occurs and is not modified until return from the function—at which time the stack frame is destroyed. We say that a program P is "well-behaved" on an input I if the execution $P(I)$ conforms strictly to this expectation, i.e., that return address fields in stack frames are modified *only* in this fashion, and the stack frames are created only when function calls are made by the program P.

DEFINITION 4.8 (WELL-BEHAVED EXECUTIONS). *Program P is* well-behaved *on input I if execution $P(I)$ maintains a call stack consisting of stack frames, one per active function call, and such that the return address in each stack frame is not modified while the corresponding function call is active.* □

Of course, a common method of exploiting a vulnerable program P involves running P on an input I' for which it is not well-behaved, i.e., that modifies a return address on the stack when the function call is still active.

The anomaly detector that we describe in this paper is assumed to be trained on the observed behaviors (emitted system calls) in executions $P(I_1), \ldots, P(I_k)$ where P is well-behaved on each I_j. In this way, the return addresses extracted from the stack (as in [3]) reflect the execution of the program. We denote these executions $P(\mathcal{I}) = \{P(I_1), \ldots, P(I_k)\}$.

4.3 Properties of Execution Graphs

Recall that an execution graph is a model constructed by a gray-box technique. None of the previous gray-box techniques, to our knowledge, has been formally related to the control flow graph of the underlying program. The execution graph differs from these approaches in the sense that the language accepted by an execution graph can be directly related to the language accepted by the control flow graph of the underlying program. Moreover, this relationship can be proved analytically. This is a significant improvement since goals of many white-box techniques can now be achieved using gray-box techniques, i.e., without static analysis on the source code or binary.

Here we show two theorems of the execution graph and the control flow graph of a program. Without loss of generality, we assume that the label of a call node in the control flow graph is the address of the instruction that immediately follows the function call or system call, which is easily obtained by static analysis of the binary. If this is not the case, e.g., if static analysis is applied on the source code, there is always a one-to-one mapping between the labels and these addresses. For convenience, we omit this mapping in the following theorems and the proofs in the Appendices.

THEOREM 4.1. *If P is a program that is well-behaved on input $\mathcal{I}$, then $L_{\text{EG}(P(\mathcal{I}))} \subseteq L_{\text{CFG}(P)}$.*

The proof of this theorem is in Appendix A.

Theorem 4.1 says that the language accepted by an execution graph is a subset of the language accepted by the control flow graph of the program, which is a property unavailable in most other gray-box techniques. It provides another level of confidence: if some execution is allowed by an execution graph, it is guaranteed that the execution is not only normal ("similar" to past executions), but also valid (allowed by the control flow graph). Such a property could only be achieved previously by white-box techniques.

Theorem 4.1 only says that $L_{\text{EG}(P(\mathcal{I}))} \subseteq L_{\text{CFG}(P)}$. They are not equal because, e.g., the input $\mathcal{I}$ might not cover all possible executions of the program, in which case there is no way for $\text{EG}(P(\mathcal{I}))$ to safely accept such a missing execution, even with the inductive definition in Definition 3.2.

Theorem 4.2 shows that if the execution graph were to be extended to allow any additional strings in the language, it could accept some intrusions that program P does not allow.

THEOREM 4.2. *Let $\mathcal{I}$ be a set of inputs, and $\text{EG}(P(\mathcal{I}))$ be an execution graph where P is well-behaved on $\mathcal{I}$. There exists a program P', which is also well-behaved on $\mathcal{I}$, such that $L_{\text{CFG}(P')} = L_{\text{EG}(P(\mathcal{I}))}$.*

Theorem 4.2 states that for any input $\mathcal{I}$ and the execution graph obtained on input $\mathcal{I}$, there exists a program P' which is well-behaved on $\mathcal{I}$, such that the language accepted by the control flow graph of this program is the same as the language accepted by the execution graph. This means that the execution graph is the "accurate" model of some program P'. Since there exists such a program P', if the execution graph were to be extended to accept any additional string in its language, it will allow an intrusion to the program P'. Informally, this means that the execution graph is a maximal graph given the set of input.

Please refer to Appendix B for a proof of Theorem 4.2.

5. PERFORMANCE EVALUATION

In this section we provide insight into the likely performance of our technique in an anomaly detection system. During program monitoring there are two tasks the anomaly detector needs to perform for each system call: (i) to walk through the stack frames and obtain all return addresses; (ii) to determine whether the current system call is allowed. We previously measured the cost of extracting program return addresses and found that for a Linux kernel compilation it adds less than 6% to the overall execution time. Therefore, extracting return addresses from the running process should introduce only moderate overhead.

Second, we measure the time it takes to process system calls when using our execution graph model. We observe the executions of four common FTP and HTTP server programs, `wu-ftpd`, `proftpd`, `Apache httpd`, and `Apache httpd` with a `chroot` patch, and extract the execution graphs from them. Information, including return addresses, of every system call is recorded into log files, and subsequently processed to detect anomalies. We measure the time it takes to process these system calls by running the anomaly detector on a desktop computer with an Intel Pentium IV 2.2 GHz CPU. Results are shown in Table 1.

Although the average processing time per system call is very different for these four programs (due to the different number of functions in the program and consequently the different number of return addresses to be processed for each system call), results show that program monitoring is extremely efficient when using the execution graph model.

6. CONCLUSION

We introduce a new model of system call behavior for anomaly detection systems, called an execution graph. Execution graph is the first gray-box model that conforms to the control flow graph of the program. We show that: an execution graph accepts only strings (defined as sequences of system calls and the active function calls when each system call occurs) that are consistent with the control flow graph of the program; and it is maximal given a set of training data, i.e., any extensions to the execution graph might make some intrusions undetected. Finally, we provide evidence that program monitoring using the execution graph is very efficient.

7. REFERENCES

[1] C. Collberg, C. Thomborson and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1998.

[2] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[3] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 62–75, May 2003.

[4] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[5] D. Gao, M. K. Reiter and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, August 2004.

[6] J. Giffin, S. Jha and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[7] J. Giffin, S. Jha and B. Miller. Efficient context-sensitive intrusion detection. In *Proceeding of*

	`wu-ftpd`	`proftpd`	`Apache`	`Apache` with `chroot` patch
number of syscalls processed	4202602	9062102	5142088	4693300
processing time test 1	0.55 s	9.63 s	2.16 s	2.92 s
processing time test 2	0.55 s	9.63 s	2.13 s	2.93 s
processing time test 3	0.54 s	9.65 s	2.15 s	2.94 s
average processing time per syscall	0.130 μs	1.063 μs	0.417 μs	0.624 μs

Table 1: Performance overhead for processing system calls

Symposium on Network and Distributed System Security, Febuary 2004.

[8] C. Kruegel, D. Mutz, F. Valeur and G. Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, October 2003.

[9] X. Lu. A Linux executable editing library. Master's Thesis, Computer and Information Science, National Unviersity of Singpaore. 1999.

[10] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, June 2003.

[11] N. Provos. Improving host security with system call policies. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.

[12] N. Provos, M. Friedl and P. Honeyman. Preventing privilege escalation. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.

[13] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad and B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proceeding of the USENIX Windows NT workshop*, August 1997.

[14] B. Schwarz, S. Debray and G. Andrews. Disassembly of executable code revisited. In *Proceeding of Working Conference on Reverse Engineering*, pages 45–54, October 2002.

[15] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, May 2001.

[16] K. Tan and R. Maxion. "Why 6?"—Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188-201, May 2002.

[17] D. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, Department of Computer Science, University of California at Berkeley, August 1999.

[18] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168, May 2001.

[19] R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham. Efficient software-based fault isolation. In *Proceeding of the Symposium on Operating System Principles*, 1993.

[20] A. Wespi, M. Dacier and H. Debar. An intrusion-detection system based on the Teiresias pattern-discovery algorithm. In *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Conference*, 1999.

[21] A. Wespi, M. Dacier and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, pages 110–129, October 2000.

APPENDIX

A. PROOF OF THEOREM 4.1

We first prove the following lemmas. As stated in Section 4.3, without loss of generality, we assume that the label of any call node in the control flow graph is in fact the address of the instruction that immediately follows the call. If this is not the case, e.g., if the control flow graph is obtained by static analysis of the source code, there is always a one-to-one mapping between the labels and these addresses. For convenience, we omit this mapping in the following proofs.

In the following proofs, we use μ to denote the length of a function call or system call instruction. Since we assume that the label of any call node x in the control flow graph is the address of the instruction that immediately follows the call, $x - \mu$ represents the address of the corresponding call instruction. We use F_v to denote the function in P that consists of node v.

LEMMA A.1. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* $\text{EG}(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* $\text{CFG}(P) = (V', E')$*, then* $V \subseteq V'$.

PROOF.
$v \in V \wedge v$ is a leaf node
$\Rightarrow$ P is able to make a system call with system call number v
$\Rightarrow$ $v \in V'$

$v \in V \wedge v$ is not a leaf node
$\Rightarrow$ v is one of the return addresses observed when P makes a system call
$\Rightarrow$ $(v - \mu)$ is the address of a call instruction
$\Rightarrow$ $(v - \mu)$ corresponds to some function or system call site in P
$\Rightarrow$ $v \in V'$
$\square$

Notice that there could be $v' \notin V$ while $v' \in V'$, because input $\mathcal{I}$ does not necessarily cover all possible executions of P, and that some executions allowed by $\text{CFG}(P)$ might never appear in actual runs.

LEMMA A.2. *Let P be a program that is well-behaved on input $\mathcal{I}$. If $\langle r_1, r_2, \ldots, r_l, \ldots\rangle$ and $\langle r'_1, r'_2, \ldots, r'_l, \ldots\rangle$ are two observations in $P(\mathcal{I})$, such that for each $1 \le i < l$, $r_i = r'_i$, $r_l \neq r'_l$, then for some function $F \in P$, r_l and r'_l are both in* $\text{CFSG}(F)$.

PROOF.

$\langle r_1, r_2, \ldots, r_l, \ldots\rangle$ and $\langle r'_1, r'_2, \ldots, r'_l, \ldots\rangle$ are two observations

$\Rightarrow$ $(r_l - \mu)$ is in a function that is called from $(r_{l-1} - \mu)$; $(r'_l - \mu)$ is in a function that is called from $(r'_{l-1} - \mu)$

Now, for each $1 \le i < l$, $r_i = r'_i$, $r_l \neq r'_l$

$\Rightarrow$ $(r_{l-1} - \mu) = (r'_{l-1} - \mu)$

$\Rightarrow$ $(r_l - \mu)$ and $(r'_l - \mu)$ are in the same function (instruction at address $(r_{l-1} - \mu)$ can call only one function)

$\Rightarrow$ r_l and r'_l are nodes in the same CFSG

□

LEMMA A.3. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $(r, r') \in E_{\mathsf{crs}}$, then there exist a sequence of nodes $\langle v_1, v_2, \ldots, v_n\rangle$ in* CFG(P) *such that*

- *$v_1 = r$, $v_n = r'$; and*
- *For each $1 \le i < n$, $(v_i, v_{i+1}) \in E'$; and*
- *For each $1 < i < n$, v_i is not a system call node; and*
- *If $n > 2$, then $\langle v_2, v_3, \ldots, v_{n-1}\rangle$ is a (series of) call cycle(s).*

PROOF.

$(r, r') \in E_{\mathsf{crs}}$

$\Rightarrow$ there exists two consecutive observations o and o' in $P(\mathcal{I})$, where $o = \langle r_1, r_2, \ldots, r_l, \ldots r_n\rangle$, $o' = \langle r'_1, r'_2, \ldots, r'_l, \ldots r'_{n'}\rangle$, such that for each $1 \le i < l$, $r_i = r'_i$, and $r_l = r$ and $r'_l = r'$ (Definition 3.2)

o and o' are two consecutive observations

$\Rightarrow$ there must be a path in CFG(P) from r_{n-1} to $r'_{n'-1}$ via r_l and r'_l that does not consist of any other system call nodes

For each $1 \le i < l$, $r_i = r'_i$, $r_l \neq r'_l$

$\Rightarrow$ r_l and r'_l are addresses in the same function (Lemma A.2)

$\Rightarrow$ any function call nodes on the path from r to r' must form a (series of) call cycles (completed function calls)

$\Rightarrow$ there must be a path in CFG(P) from r to r' that satisfies the claimed properties.

□

LEMMA A.4. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $(r, r') \in E_{\mathsf{call}}$ and r' is not a leaf node, then there exists a sequence of nodes $\langle v_1, v_2, \ldots, v_n\rangle$ in* CFG(P) *such that*

- *$v_1 = r$, $v_2 = F_{r'}.\mathsf{enter}$, $v_n = r'$; and*
- *For each $1 \le i < n$, $(v_i, v_{i+1}) \in E'$; and*
- *For each $3 \le i < n$, v_i is not a system call node; and*
- *If $n > 3$, then $\langle v_3, v_4, \ldots, v_{n-1}\rangle$ is a (series of) call cycle(s).*

PROOF. According to Definition 3.2, $(r, r') \in E_{\mathsf{call}}$ results from at least one of the following three conditions. We prove Lemma A.4 in all these three conditions.

- (Base case of Definition 3.2)

  There exists two consecutive observations o and o' in $P(\mathcal{I})$, where $o = \langle r_1, r_2, \ldots, r_l, \ldots\rangle$, $o' = \langle r'_1, r'_2, \ldots, r'_l, \ldots, r, r', \ldots\rangle$, and for each $1 \le i < l$, $r_i = r'_i$ and $(r_l, r'_l) \in E_{\mathsf{crs}}$

  $\Rightarrow$ after the system call corresponding to o is executed, execution has to return to CFSG(F_{r_l}) and then follow the path as described in Lemma A.3 and subsequently enter CFSG(F_r) and CFSG$(F_{r'})$ in order to make system call that corresponds to o'

  $\Rightarrow$ instruction at $(r - \mu)$ calls function $F_{r'}$, and there must be a path in CFG(P) from $F_{r'}.\mathsf{enter}$ to r' that satisfies the claimed properties.

- (First induction of Definition 3.2) Given $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \overset{\mathsf{crs}}{\rightarrow} x_2$, $(x_2, x_3) \in E_{\mathsf{rtn}}$, $x_3 = r$ and $x_1 = r'$

  $(x_0, x_1) \in E_{\mathsf{call}}$

  $\Rightarrow$ there exists a path in CFG(P) from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties. (Base case in this proof)

  Since we have already found the path from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties, it only remains to prove that $(x_3, F_{x_1}.\mathsf{enter}) \in E'$.

  $x_1 \overset{\mathsf{crs}}{\rightarrow} x_2$

  $\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma A.2)

  $(x_2, x_3) \in E_{\mathsf{rtn}}$

  $\Rightarrow$ $(F_{x_2}.\mathsf{exit}, x_3) \in E'$

  $\Rightarrow$ $(x_3, F_{x_2}.\mathsf{enter}) \in E'$

  $\Rightarrow$ $(x_3, F_{x_1}.\mathsf{enter}) \in E'$

- (Second induction of Definition 3.2) Given $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \overset{\mathsf{crs}}{\rightarrow} x_2$, and $(x_3, x_2) \in E_{\mathsf{call}}$,

  – When $x_3 = r$ and $x_1 = r'$

    $(x_0, x_1) \in E_{\mathsf{call}}$

    $\Rightarrow$ there exists a path in CFG(P) from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties. (Base case in this proof)

    Since we have already found the path from $F_{x_1}.\mathsf{enter}$ to x_1 satisfying the claimed properties, it only remains to prove that $(x_3,\ F_{x_1}.\mathsf{enter}) \in E'$.

    $x_1 \overset{\mathsf{crs}}{\rightarrow} x_2$

    $\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma A.2)

    $(x_3, x_2) \in E_{\mathsf{call}}$

    $\Rightarrow$ $(x_3, F_{x_2}.\mathsf{enter}) \in E'$

    $\Rightarrow$ $(x_3, F_{x_1}.\mathsf{enter}) \in E'$

  – When $x_0 = r$ and $x_2 = r'$

    $(x_3, x_2) \in E_{\mathsf{call}}$

    $\Rightarrow$ there exists a path in CFG(P) from $F_{x_2}.\mathsf{enter}$ to x_2 that satisfies the claimed properties. (Base case in this proof)

    Since we have already found the path from $F_{x_2}.\mathsf{enter}$ to x_2 satisfying the claimed properties, it only remains to prove that $(x_0,\ F_{x_2}.\mathsf{enter}) \in E'$.

    $x_1 \overset{\mathsf{crs}}{\rightarrow} x_2$

    $\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma A.2)

    $(x_0, x_1) \in E_{\mathsf{call}}$

    $\Rightarrow$ $(x_0, F_{x_1}.\mathsf{enter}) \in E'$

    $\Rightarrow$ $(x_0, F_{x_2}.\mathsf{enter}) \in E'$

□

Analagous to Lemma A.4, we have Lemma A.5 as shown below (proof of which is skipped).

LEMMA A.5. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $(r, r') \in E_{\mathsf{rtn}}$ and r is not a leaf node, then there exists a sequence of nodes $\langle v_1, v_2, \ldots, v_n\rangle$ in* CFG(P) *such that*

- $v_1 = r$, $v_{n-1} = F_r.\mathsf{exit}$, $v_n = r'$; *and*
- *For each* $1 \le i < n$, $(v_i, v_{i+1}) \in E'$; *and*
- *For each* $1 < i \le n-3$, v_i *is not a system call node; and*
- *If* $n > 3$, *then* $\langle v_2, v_3, \ldots, v_{n-2}\rangle$ *is a (series of) call cycle(s).*

LEMMA A.6. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$. *If $\langle r_1, r_2, \ldots, r_n\rangle$ is an execution stack, then there exists an observable path π in* CFG(P) *such that* $A(\pi) = \langle r_1,\ r_2,\ \ldots,\ r_n\rangle$.

PROOF.
$\langle r_1, r_2, \ldots, r_n\rangle$ is an execution stack
$\Rightarrow$ for each $1 \le i < n$, $r_i \xrightarrow{\mathsf{xcall}} r_{i+1}$ (Definition 3.4)
$\Rightarrow$ for each $1 \le i < n$, $(r_i, r_{i+1}) \in E_{\mathsf{call}}$ or $\left(\exists z : (r_i, z) \in E_{\mathsf{call}} \wedge z \xrightarrow{\mathsf{crs}} r_{i+1}\right)$ (Definition 3.4)

If for any $1 \le i < n-1$, $(r_i, r_{i+1}) \in E_{\mathsf{call}}$
$\Rightarrow$ there exists a path $\langle r_i, F_{r_{i+1}}.\mathsf{enter}, \ldots, r_{i+1}\rangle$ in CFG(P) (Lemma A.4).

If for any $1 \le i < n-1$, $(r_i, z) \in E_{\mathsf{call}} \wedge z \xrightarrow{\mathsf{crs}} r_{i+1}$
$\Rightarrow$ there exists a path $\langle r_i,\ F_{r_{i+1}}.\mathsf{enter},\ \ldots,\ z,\ \ldots,\ r_{i+1}\rangle$ in CFG(P) (Lemma A.4 and Lemma A.3).

Connecting $\langle \mathtt{main}.\mathsf{enter}, \ldots, r_1\rangle$ and all these paths from each r_i to r_{i+1} together forms an observable path π that traverses $r_1, r_2, \ldots, r_{n-1}$.

Since each individual path $\langle r_i, \ldots, r_{i+1}\rangle$ consists of only $r_i, F_{r_{i+1}}.\mathsf{enter}, r_{i+1}$ and a (possibly empty series of) call cycle(s) (Lemma A.4), $A(\pi) = \langle r_1, r_2, \ldots, r_n\rangle$ (Definition 4.6). □

LEMMA A.7. *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $s = \langle r_1, r_2, \ldots, r_m\rangle$ and $s' = \langle r'_1, r'_2, \ldots, r'_{m'}\rangle$ are execution stacks in V, and s' is a successor of s, then there exist a sequence of nodes $\langle v_1, v_2, \ldots, v_n\rangle$ in* CFG(P) *such that*

- $v_1 = r_{m-1}$, $v_n = r'_{m'-1}$; *and*
- *For each* $1 \le i < n$, $(v_i, v_{i+1}) \in E'$; *and*
- *For each* $1 < i < n$, v_i *is not a system call node; and*
- *Let* $\langle v_{l_1}, v_{l_2}, \ldots, v_{l_j}\rangle$ *denote the remaining sequence of nodes when all call cycles on* $\langle v_1, v_2, \ldots, v_n\rangle$ *are removed, where for each* $1 \le i < j$, $l_i < l_{i+1}$. *Then there exists an integer k, such that*
  - *For each* $1 \le i < k$, $r_i = r'_i$; *and*
  - $j = 2(m + m' - 2k - 1)$; *and*
  - $v_{l_1} = r_{m-1}$, $v_{l_2} = F_{r_{m-1}}.\mathsf{exit}$;
    $\ldots$;
    $v_{l_{2(m-k)-3}} = r_{k+1}$, $v_{l_{2(m-k)-2}} = F_{r_{k+1}}.\mathsf{exit}$;
    $v_{l_{2(m-k)-1}} = r_k$, $v_{l_{2(m-k)}} = r'_k$;
    $v_{l_{2(m-k)+1}} = F_{r'_{k+1}}.\mathsf{enter}$, $v_{l_{2(m-k)+2}} = r'_{k+1}$;
    $\ldots$;
    $v_{l_{2(m+m'-2k)-3}} = F_{r'_{m'-1}}.\mathsf{enter}$, $v_{l_{2(m+m'-2k)-2}} = r'_{m'-1}$.

PROOF.
s' is a successor of s
$\Rightarrow$ there exists an integer k such that $r_m \xrightarrow{\mathsf{rtn}} r_k$, $(r_k, r'_k) \in E_{\mathsf{crs}}$, $r'_k \xrightarrow{\mathsf{call}} r'_{m'}$ and for each $1 \le i < k$, $r_i = r'_i$ (Definition 3.5)
$\Rightarrow$ there exist three paths from r_{m-1} to r_k (Lemma A.5), from r_k to r'_k (Lemma A.3) and from r'_k to $r'_{m'-1}$ (Lemma A.4)
$\Rightarrow$ connecting the above 3 paths together forms the sequence of nodes with the claimed properties.
□

LEMMA A.8. *Let P be a program that is well-behaved on input $\mathcal{I}$. If there is an execution path $\delta = \langle s_1, \ldots, s_n\rangle$ in* EG$(P(\mathcal{I}))$, *where $s_i = \langle r_{i,1}, \ldots, r_{i,m_i}\rangle$, then there exists an observable path* $\pi = \langle \mathtt{main}.\mathsf{enter}, \ldots, r_{1,m_1-1}, \ldots, r_{2,m_2-1}, \ldots, r_{n,m_n-1}\rangle$ *in* CFG(P) *such that*

- $r_{1,m_1-1}, r_{2,m_2-1}, \ldots, r_{n,m_n-1}$ *are the only system call nodes on π; and*
- *Let* $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n\rangle$, *then for each* $1 \le i \le n$, $A(\pi_i) = s_i$.

PROOF. According to Lemma A.6, there exists a path $\beta_0 = \langle \mathtt{main}.\mathsf{enter},\ \ldots,\ r_{1,1},\ F_{r_{1,2}}.\mathsf{enter},\ \ldots,\ r_{1,2},\ \ldots,\ r_{1,m_1-1}\rangle$ in CFG(P). Since for each $1 \le i < m_1$, $(r_{1,i},\ r_{1,i+1}) \in E_{\mathsf{call}}$ (Definition 3.6), r_{1,m_1-1} is the only system call node on β_0 (Lemma A.4).

According to Lemma A.7, for each $1 \le i < n$, there is a path $\beta_i = \langle r_{i,m_i-1},\ \ldots,\ F_{r_{i,m_i-1}}.\mathsf{exit},\ \ldots,\ F_{r_{i,k_i+1}}.\mathsf{exit},\ r_{i,k_i},\ \ldots,\ r_{i+1,k_i},\ F_{r_{i+1,k_i+1}}.\mathsf{enter},\ \ldots,\ F_{r_{i+1,m_{i+1}-1}}.\mathsf{enter},\ \ldots,\ r_{i+1,m_{i+1}-1}\rangle$, where for each $1 \le j < k_i$, $r_{i,j} = r_{i+1,j}$.

Connecting $\beta_0, \beta_1, \ldots, \beta_{n-1}$ together forms π.

Since the only system call nodes on β_i are r_{i,m_i-1} and $r_{i+1,m_{i+1}-1}$ (Lemma A.7), $r_{1,m_1-1}, r_{2,m_2-1}, \ldots, r_{n,m_n-1}$ are the only system call nodes on π.

Let $k_0 = m_1 - 1$ and for each $1 \le i < n$, let $l_i = \min(k_{i-1}, k_i)$. Since the subsequence $\langle r_{i,l_i},\ F_{r_{i,l_i+1}}.\mathsf{enter},\ \ldots,\ F_{r_{i,m_i-1}}.\mathsf{enter},\ \ldots,\ r_{i,m_i-1}\rangle$ from β_{i-1} and $\langle r_{i,m_i-1},\ \ldots,\ F_{r_{i,m_i-1}}.\mathsf{exit},\ \ldots,\ F_{r_{i,l_i+1}}.\mathsf{exit},\ r_{i,l_i}\rangle$ from β_i forms a (possibly empty) call cycle, they will be deleted from $A(\pi)$ (First step of the procedure in Definition 4.6).

Since for each $1 \le i < n$ and each $1 \le j < k_i$, $r_{i,j} = r_{i+1,j}$ (Definition 3.5), the remaining nodes of $A(\pi)$ after the second step of the procedure in Definition 4.6 are $\langle r_{n,1},\ r_{n,2},\ \ldots,\ r_{n,m_n-1}\rangle$. Therefore, $A(\pi) = s_n$.

$A(\pi_i) = s_i$ can be proved similarly. □

Theorem 4.1 follows immediately from Lemma A.8.

B. PROOF OF THEOREM 4.2

To show the existence of such a program P', we (i) build a graph G' from the execution graph EG$(P(\mathcal{I}))$; (ii) show that G' is the control flow graph of some program P' that is well-behaved on input $\mathcal{I}$, i.e., CFG$(P') = G'$; and (iii) show that $L_{\text{CFG}(P')} = L_{\text{EG}(P(\mathcal{I}))}$.

DEFINITION B.1 (*E2G*). *The operation E2G takes as input an execution graph* EG$(P(\mathcal{I})) = (V,\ E_{\mathsf{call}},\ E_{\mathsf{crs}},\ E_{\mathsf{rtn}})$ *and performs the following operations:*

1. *For each $x_i \in V$ where*
   - *x_i is not a leaf node; and*
   - *there does not exist a leaf node v such that $(x_i, v) \in E_{\mathsf{call}}$.*

   Let

$$\begin{aligned}
C(x_i) &= \{v : (x_i, v) \in E_{\mathsf{call}}\} \\
C'(x_i) &= \left\{v : [\exists v' \in C(x_i) : v' \overset{\mathsf{crs}}{\to} v]\right\} \\
C''(x_i) &= \{v : v \textit{ is a leaf node} \wedge \\
&\quad [\exists v' \in (C(x_i) \cup C'(x_i)) : (v', v) \in E_{\mathsf{call}}]\} \\
E(x_i) &= \{(v_1, v_2) : v_1 \in (C(x_i) \cup C'(x_i)) \wedge \\
&\quad v_2 \in (C(x_i) \cup C'(x_i)) \wedge (v_1, v_2) \in E_{\mathsf{crs}}\} \\
E'(x_i) &= \{(v_1, v_2) : v_1 \in (C(x_i) \cup C'(x_i)) \wedge \\
&\quad v_2 \in C''(x_i) \wedge (v_1, v_2) \in E_{\mathsf{call}}\}
\end{aligned}$$

2. *Define the equivalence relation $x_i \sim x_j$ if $C(x_i) = C(x_j)$. Let $[x_i]$ denote the equivalence class of x_i. For each equivalence class $[x_i]$, let $G_{[x_i]} = (V_{[x_i]}, E_{[x_i]})$ where*

$$\begin{aligned}
V_{[x_i]} &= C(x_i) \cup C'(x_i) \cup C''(x_i) \cup \\
&\quad \{G_{[x_i]}.\mathsf{enter}, G_{[x_i]}.\mathsf{exit}\} \\
E_{[x_i]} &= E(x_i) \cup E'(x_i) \cup \\
&\quad \{(G_{[x_i]}.\mathsf{enter}, v) : v \in C(x_i)\} \cup \\
&\quad \{(v, G_{[x_i]}.\mathsf{exit}) : v \in V_{[x_i]} \wedge (v, x_i) \in E_{\mathsf{rtn}}\}
\end{aligned}$$

3. *Create a new graph $G' = (V', E')$, such that*

$$\begin{aligned}
V' &= \left(\bigcup_{[x_i]} V_{[x_i]}\right) \cup M \\
E' &= \bigcup_{[x_i]} \big(E_{[x_i]} \cup \{(v, G_{[x_i]}.\mathsf{enter}) : v \in [x_i]\} \cup \\
&\quad \{(G_{[x_i]}.\mathsf{exit}, v) : v \in [x_i]\}\big) \\
&\quad \cup \{(v_1, v_2) : \{v_1, v_2\} \subseteq M \wedge (v_1, v_2) \in E_{\mathsf{crs}}\}
\end{aligned}$$

   where

$$\begin{aligned}
M &= \{v : v \in V \wedge \\
&\quad [\textit{there does not exist } v' \in V : v' \overset{\mathsf{xcall}}{\to} v]\}
\end{aligned}$$

Operation E2G returns the graph G'. □

In the above definition, M is the set of nodes that represent addresses in `main()`. With Definition B.1, we are done with the first step in our proof. The next step is to prove that CFG$(P') = G'$ for some program P' that is also well-behaved on input $\mathcal{I}$.

LEMMA B.1. *If graph $G' = (V', E')$ is the output of operation E2G on an execution graph* EG$(P(\mathcal{I}))$*, then there exists some program P' which is well-behaved on input $\mathcal{I}$, such that* CFG$(P') = G'$.

Though we do not provide the proof of Lemma B.1, the following is the intuition. From Definition B.1, one can notice that graph G' contains a set of subgraphs, which are connected by directed edges from a function call node to the entry node of the function subgraph, and from the exit node of the function subgraph to the same function call node. Besides that, each subgraph contains function call nodes and system call nodes, as well as one entry node and one exit node. When given this graph G', programming languages such as C and C++ can be used to implement each subgraph as a function, and implement the entire graph G' as a program P'. If implemented correctly, the implementation output P' will be well-behaved on input $\mathcal{I}$, and the control flow graph of P' will be the same as G'.

The last step in our proof of Theorem 4.2 is to show that $L_{\text{CFG}(P')} = L_{\text{EG}(P(\mathcal{I}))}$, where CFG$(P') = G' = (V', E')$. To prove this we need to show that (i) $L_{\text{EG}(P(\mathcal{I}))} \subseteq L_{\text{CFG}(P')}$, and (ii) $L_{\text{CFG}(P')} \subseteq L_{\text{EG}(P(\mathcal{I}))}$. The proof of (i) is very similar to the proof of Theorem 4.1 and it is skipped in this paper. We only show the important lemmas for the proof of (ii). Notice that the difference between these two proofs and those in Appendix A is that here V' and E' are given as in Definition B.1, whereas in Theorem 4.1 they are not given.

LEMMA B.2. *Let P be a program that is well-behaved on input $\mathcal{I}$, and $E2G$(*EG$(P(\mathcal{I}))$*) $= G'$. If π is an observable path in G', then there exists an execution stack s in* EG$(P(\mathcal{I}))$ *such that $s = A(\pi)$.*

LEMMA B.3. *Let P be a program that is well-behaved on input $\mathcal{I}$, and $E2G$(*EG$(P(\mathcal{I}))$*) $= G'$. Let π be an observable path in G', and $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$, then $\langle A(\pi_1), A(\pi_2), \ldots, A(\pi_n) \rangle$ is an execution path in* EG$(P(\mathcal{I}))$.