Automatically Adapting a Trained Anomaly Detector to Software Patches

Peng Li[1], Debin Gao[2], and Michael K. Reiter[1]

[1] Department of Computer Science, University of North Carolina, Chapel Hill, NC, USA
[2] School of Information Systems, Singapore Management University, Singapore

Abstract. In order to detect a compromise of a running process based on it deviating from its program's normal system-call behavior, an anomaly detector must first be trained with traces of system calls made by the program when provided clean inputs. When a patch for the monitored program is released, however, the system call behavior of the new version might differ from that of the version it replaces, rendering the anomaly detector too inaccurate for monitoring the new version. In this paper we explore an alternative to collecting traces of the new program version in a clean environment (which may take effort to set up), namely adapting the anomaly detector to accommodate the differences between the old and new program versions. We demonstrate that this adaptation is feasible for such an anomaly detector, given the output of a state-of-the-art binary difference analyzer. Our analysis includes both proofs of properties of the adapted detector, and empirical evaluation of adapted detectors based on four software case studies.

Keywords: Anomaly detection, software patches, system-call monitoring, binary difference analysis

1 Introduction

One widely studied avenue for detecting the compromise of a process (e.g., by a buffer overflow exploit) is by monitoring its system-call behavior. So-called "white-box" detectors build a model of system-call behavior for the program via static analysis of the source code or binary (e.g., [18, 5, 11, 12, 2, 13]). "Black-box" (or "gray-box") detectors are trained with system-call traces of the program when processing intended inputs (e.g., [7, 6, 15, 16, 9, 8]). In either case, deviation of system-call behavior from the model results in an alarm being raised, as this might indicate that the code executing in the process has changed. Both white-box and black/gray-box approaches offer advantages. The hallmark of white-box approaches is the potential for a near-zero or zero false alarm rate [18], if static analysis uncovers every possible system call sequence that the program could possibly emit. Since they are trained on "normal" system-call behavior, black/gray-box approaches can be more sensitive, in that they can reflect nuances of the local environments and usage of the monitored programs [14] and can detect behavioral anomalies that are nevertheless consistent with the control-flow graph of the program. Such anomalies can indicate a compromise (e.g., [3]) and, if ignored, allow more room for mimicry attacks to succeed [19, 17].

When a monitored program is patched, an anomaly detector trained on system-call traces may no longer be sufficiently accurate to monitor the updated program. One way to address this is to rebuild the model by collecting traces of the updated program. However, these traces must be gathered in a sanitized environment free of attacks that is otherwise as similar as possible — e.g., in terms of the operating system and relevant device configurations and contents, as well as the program usage — to the environment in which the updated program will be run. This problem is compounded if there are multiple such environments.

To avoid the effort of setting up a sanitized environment for collecting system-call traces every time a patch is issued, in this paper we consider an alternative approach to building a model of normal system-call behavior for an updated program. Our approach consists of detecting the differences between the updated program and the previous version, and then directly updating the system-call behavior model to reflect these changes. There are several complexities that arise in doing this, however. First, program patches are often released as wholly new program versions, not isolated patches. Second, in either case, program updates are typically released only in binary format. Both of these make it difficult to detect where the changes occur between versions. Third, while state-of-the-art binary difference analyzers (e.g., [10]) can detect where changes occur, how to modify the system-call model to reflect those changes can require significant further analysis. We emphasize, in particular, that we would like to adapt the model to accommodate these changes while decaying the model's sensitivity to abnormal behavior as little as possible. So, adaptations that increase the model's size (and hence allowed behaviors) more than the changes would warrant should be avoided.

In this paper we provide an algorithm for converting the *execution-graph* anomaly detector [8] on the basis of the output of the BinHunt binary difference analysis tool [10] when applied to a program and its updated version. We show that our algorithm is sound, in the sense that the resulting execution-graph anomaly detector accepts only system-call sequences that are consistent with the control-flow graph of the program. Such soundness was also a requirement of the original execution-graph model [8], and so our algorithm preserves this property of the converted execution graph. In addition, we show through experiments with several patched binaries that our converted execution graphs can be of comparable size to ones generated by training on system-call sequences collected from the updated program, and moreover that the converted execution graphs accept (i.e., do not raise alarms on) those sequences. As such, the converted execution graphs from our algorithms are, based on our experiments, good approximations of the execution graphs that would have been achieved by training. To our knowledge, ours is the first work to automatically update a system-call-based anomaly detection model in response to program patches.

2 Related work

Systems that employ binary matching techniques to reuse stale "profiles" are most related to our work. Profiles of a program are representatives of how a program is used on a specific machine by a specific user. They usually include program counter information, memory usage, system clock information, etc., and are typically obtained by

executing an instrumented version of the program that generates profile information as a side-effect of the program execution. Spike [4] is an optimization system that collects, manages, and applies profile information to optimize the execution of DEC Alpha executables. When old profiles are used to optimize a new build of a program, Spike simply discards profiles for procedures that have changed, where changes in procedures between two builds of a program are detected by calculating the edit distance between signatures of the corresponding procedures. Spike is not able to re-use profiles of modified procedures.

Wang et al. proposed a binary matching tool, namely BMAT, to propagate profile information from an older, extensively profiled build to a newer build [20]. An optimized version of the newer build is then obtained by applying optimization techniques on the newer build and the propagated profile. The main difference between BMAT and our proposed technique is that we skip the process of propagating the profiles (which roughly correspond to the system-call traces in anomaly detection) and directly propagate the anomaly detection model of the older build to that of the newer build. Our approach is better suited to anomaly detectors that use an automaton-like model because these models are closely related to the control flow of the program (e.g., [8]), and therefore our approach avoids potential inaccuracies introduced in an indirect approach in which system-call traces are derived first.

3 Background and terminology

To better explain our algorithm for converting the execution-graph anomaly detection model [8], here we provide some background and terminology. We first give our definitions of basic blocks and control flow graphs, which are slightly different from those typical in the literature (c.f., [1]). Next, we outline important concepts in binary difference analysis including common induced subgraphs and relations between two matched basic blocks and two matched functions. We also define important elements in control flow graphs, e.g., call cycles and paths, and finally briefly define an execution graph. The conversion algorithms and their properties presented in Section 4 rely heavily on the definitions and lemmas outlined in this section.

Our definitions below assume that each function is entered only by calling it; jumping into the middle of a function (e.g., using a `goto`) is presumed not to occur. We consider two system calls the same if and only if they invoke the same system-call interface (with potentially different arguments).

Definition 1 [basic block, control-flow subgraph/graph] A *basic block* is a consecutive sequence of instructions with one entry point. The last instruction in the basic block is the first instruction encountered that is a jump, function call, or function return, or that immediately precedes a jump target.

The *control-flow subgraph* of a function f is a directed graph $\mathsf{cfsg}_f = \langle \mathsf{cfsgV}_f, \mathsf{cfsgE}_f \rangle$. cfsgV_f contains

- a designated f.`enter` node and a designated f.`exit` node; and
- a node per basic block in f. If a basic block ends in a system call or function call, then its node is a *system call node* or *function call node*, respectively. Both types of

nodes are generically referred to as simply *call nodes*. Each node is named by the address immediately following the basic block.[3]

cfsgE_f contains (v, v') if

- $v = f$.`enter` and v' denotes the first basic block executed in the function; or
- $v' = f$.`exit` and v ends with a return instruction; or
- v ends in a jump for which the first instruction of v' is the jump target; or
- the address of the first instruction of v' is the address immediately following (i.e., is the name of) v.

The *control-flow graph* of a program P is a directed graph $\mathsf{cfg}_P = \langle \mathsf{cfgV}_P, \mathsf{cfgE}_P \rangle$ where $\mathsf{cfgV}_P = \bigcup_{f \in P} \mathsf{cfsgV}_f$ and $(v, v') \in \mathsf{cfgE}_P$ iff

- $(v, v') \in \mathsf{cfsgE}_f$ for some $f \in P$; or
- $v' = f$.`enter` for some $f \in P$ and v denotes a basic block ending in a call to f; or
- $v = f$.`exit` for some $f \in P$ and v' denotes a basic block ending in a call to f.

□

We next define common induced subgraphs, which are used in binary difference analysis of two programs [10].

Definition 2 [common induced subgraph, $\sim$, $\approx$] Given $\mathsf{cfsg}_f = \langle \mathsf{cfsgV}_f, \mathsf{cfsgE}_f \rangle$, an *induced subgraph* of cfsg_f is a graph $\mathsf{isg}_f = \langle \mathsf{isgV}_f, \mathsf{isgE}_f \rangle$ where $\mathsf{isgV}_f \subseteq \mathsf{cfsgV}_f$ and $\mathsf{isgE}_f = \mathsf{cfsgE}_f \cap (\mathsf{isgV}_f \times \mathsf{isgV}_f)$. Given two functions f and g, a *common induced subgraph* is a pair $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$ of induced subgraphs of cfsg_f and cfsg_g, respectively, that are isomorphic. We use $\sim$ to denote the node isomorphism; i.e., if $v \in \mathsf{isgV}_f$ maps to $w \in \mathsf{isgV}_g$ in the isomorphism, then we write $v \sim w$ and say that v "matches" w. Similarly, if $v \sim w$, $v' \sim w'$, and $(v, v') \in \mathsf{isgE}_f$ (and so $(w, w') \in \mathsf{isgE}_g$), then we write $(v, v') \sim (w, w')$ and say that edge (v, v') "matches" (w, w').

The algorithm presented in this paper takes as input an injective partial function $\pi : \{f : f \in P\} \rightarrow \{g : g \in Q\}$ for two programs P and Q, and induced subgraphs $\{\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle : \pi(f) \neq \perp\}$. We naturally extend the "matching" relation to functions by writing $f \sim \pi(f)$ if $\pi(f) \neq \perp$, and say that f "matches" $\pi(f)$. Two matched functions f and g are *similar*, denoted $f \approx g$, iff $\mathsf{isg}_f = \mathsf{cfsg}_f$ and $\mathsf{isg}_g = \mathsf{cfsg}_g$. □

Control-flow subgraphs and graphs, and common induced subgraphs for two programs, can be extracted using static analysis of binaries [10]. When necessary, we will appeal to static analysis in the present work, assuming that static analysis is able to disassemble the binary successfully to locate the instructions in each function, and to build cfsg_f for all functions f and cfg_P for the program P.

A tool that provides the common induced subgraphs required by our algorithm is BinHunt [10]. When two nodes are found to match each other by BinHunt, they are functionally similar. For example, if $v \in \mathsf{isgV}_f$, $w \in \mathsf{isgV}_{\pi(f)}$, and $v \sim w$, then either both v and w are call nodes, or neither is; we utilize this property in our algorithm. However, BinHunt compares two nodes by analyzing the instructions *within* each node only, and so the meaning of *match* does not extend to functions called by the nodes. For example, two nodes, each of which contains a single `call` instruction, may match to each other even if they call very different functions. In order to extend the meaning

[3] For a function call node, this name is the return address for the call it makes.

of *match* to functions called by the nodes, we introduce a new relation between two functions (and subsequently two nodes), called *extended similarity*.

Definition 3 [$\overset{\text{ext}}{\approx}$] Two matched functions f and g are *extended-dissimilar*, denoted $f \overset{\text{ext}}{\not\approx} g$, iff

– (Base cases)
  - $f \not\approx g$; or
  - for two system call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, v and w call different system calls; or
  - for two function call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, if v calls f' and w calls g', then $f' \not\approx g'$.
– (Induction) For two function call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, if v calls f' and w calls g', then $f' \overset{\text{ext}}{\not\approx} g'$.

If two matched functions f and g are not extended-dissimilar, then they are *extended-similar*, denoted $f \overset{\text{ext}}{\approx} g$. Two matched nodes v and w are *extended-similar*, denoted $v \overset{\text{ext}}{\approx} w$, if (i) neither v nor w is a call node; or (ii) v and w make the same system call; or (iii) v and w call f and g, respectively, and $f \overset{\text{ext}}{\approx} g$. □

Two extended-similar nodes exhibit a useful property that will be stated in Lemma 1. To state this property, we first define call cycles.

Definition 4 [Call cycle] A sequence of nodes $\langle v_1, \ldots, v_l \rangle$ in cfg_P is a *call cycle from* v iff for some function $f \in P$, $v = v_1 = v_l$ is a function call node calling to f, $v_2 = f.\texttt{enter}$, $v_{l-1} = f.\texttt{exit}$, and

– (Base case) For each $i \in (1, l-1)$, $v_i \in \mathsf{cfsgV}_f$ and $(v_i, v_{i+1}) \in \mathsf{cfsgE}_f$.
– (Induction) For some $k, k' \in (1, l-1)$, $k < k'$,
  - for each $i \in (1, k] \cup [k', l)$, $v_i \in \mathsf{cfsgV}_f$; and
  - for each $i \in (1, k) \cup [k', l-1)$, $(v_i, v_{i+1}) \in \mathsf{cfsgE}_f$; and
  - $\langle v_k, \ldots, v_{k'} \rangle$ is a call cycle from $v_k = v_{k'}$.

□

Lemma 1 *If v and w are call nodes in P and Q, respectively, and $v \overset{\text{ext}}{\approx} w$, then for every call cycle from v that results in a (possibly empty) sequence of system calls, there is a call cycle from w that results in the same sequence of system calls.*

Lemma 1, which is proved in Appendix B, shows a useful property about extended-similar nodes, and is used in our proofs of properties of the converted execution graph. As we will see, some edges can be copied from the execution graph of the old binary P to the execution graph of the new binary Q on the basis of nodes in cfg_P being extended-similar to nodes in cfg_Q, since those nodes exhibit similar system-call behavior. Next, we define paths to help refer to sequences of nodes in a control flow graph.

Definition 5 [Path, full, pruned, silent, audible] A *path* $p = \langle v_1, \ldots, v_n \rangle$ is a sequence of nodes where

– for all $i \in [1, n]$, $v_i \in \mathsf{cfgV}_P$; and
– for all $i \in [1, n)$, $(v_i, v_{i+1}) \in \mathsf{cfgE}_P$.

We use $|p|$ to denote the length of p which is n.

p is *pruned* if no $v \in \{v_2, \ldots, v_n\}$ is a function `enter` node, and if no $v \in \{v_1, \ldots, v_{n-1}\}$ is a function `exit` node. p is *full* if for every function call node $v \notin \{v_1, v_n\}$ on p, v is either followed by a function `enter` node or preceded by a function `exit` node (but not both).

p is called *silent* if for all $i \in (1, n)$, v_i is not a system call node. Otherwise, it is called *audible*. □

Next, we define an execution graph [8], which is a model for system-call-based anomaly detection. We begin with two technical definitions, however, that simplify the description of an execution graph.

Definition 6 [Entry call node, exit call node] A node $v \in \mathsf{cfsg}_f$ is an *entry call node* of f if v is a call node and there exists a full silent path $p = \langle f.\texttt{enter}, \ldots, v \rangle$. A node $v \in \mathsf{cfsg}_f$ is an *exit call node* of f if v is a call node and there exists a full silent path $p = \langle v, \ldots, f.\texttt{exit} \rangle$. □

Definition 7 [support ($\rightsquigarrow$), strong support ($\overset{\mathsf{s}}{\rightsquigarrow}$)] A (full or pruned) path $p = \langle v, \ldots, v' \rangle$ *supports* an edge (v, v'), denoted $p \rightsquigarrow (v, v')$, if p is silent. p *strongly supports* (v, v'), denoted $p \overset{\mathsf{s}}{\rightsquigarrow} (v, v')$, if $p \rightsquigarrow (v, v')$ and if each of v and v' is a system call node or a function call node from which there is at least one audible call cycle. □

Definition 8 [Execution subgraph/graph] An *execution subgraph* of a function f is a directed graph $\mathsf{esg}_f = \langle \mathsf{esgV}_f, \mathsf{esgE}_f \rangle$ where $\mathsf{esgV}_f \subseteq \mathsf{cfsgV}_f$ consists only of call nodes. If $(v, v') \in \mathsf{esgE}_f$ then there is a full path $p = \langle v, \ldots, v' \rangle$ such that $p \overset{\mathsf{s}}{\rightsquigarrow} (v, v')$.

An *execution graph* of a program P is a directed graph $\mathsf{eg}_P = \langle \mathsf{egV}_P, \mathsf{egEcl}_P, \mathsf{egEcr}_P, \mathsf{egErt}_P \rangle$ where egEcl_P, egEcr_P, and egErt_P are sets of *call edges*, *cross edges* and *return edges*, respectively. $\mathsf{egV}_P = \bigcup_{f \in P} \mathsf{esgV}_f$ and $\mathsf{egEcr}_P = \bigcup_{f \in P} \mathsf{esgE}_f$. If $(v, v') \in \mathsf{egEcl}_P$, then v is a function call node ending in a call to the function f containing v', and v' is a entry call node. If $(v', v) \in \mathsf{egErt}_P$, then v is a function call node ending in a call to the function f containing v', and v' is an exit call node. □

An execution graph eg_P is built by subjecting P to a set of legitimate inputs in a protected environment, and recording the system calls that are emitted and the return addresses on the function call stack when each system call is made. This data enables the construction of an execution graph. Then, to monitor a process ostensibly running P in the wild, the return addresses on the stack are extracted from the process when each system call is made. The monitor determines whether the sequence of system call (and the return addresses when those calls are made) are consistent with traversal of a path in eg_P. Any such sequence is said to be in the *language accepted by the execution graph*. Analogous monitoring could be performed using cfg_P, instead, and so we can similarly define a *language accepted by the control flow graph*. An execution graph eg_P is built so that any sequence in its language is also in the language accepted by cfg_P [8].

4 The conversion algorithm

Suppose that we have an execution graph eg_P for a program P, and that a patch to P is released, yielding a new program Q. In this section, we show our conversion algorithm to obtain eg_Q. In addition to utilizing eg_P, our algorithm utilizes the output of a binary difference analysis tool (e.g., [10]), specifically a partial injective function π and pairs $\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle$ of isomorphic induced subgraphs. Our algorithm also selectively uses

static analysis on Q. Unless stated otherwise, below we use f, v and p to denote a function, node and path, respectively, in cfg_P, and we use g, w, and q to denote a function, node and path, respectively, in cfg_Q. In addition, we abuse notation in using "$\in$" to denote a path being in a graph (e.g., "$p \in \mathsf{cfg}_P$"), in addition to its normal use for set membership.

Recall that we have two important requirements in designing the conversion algorithm. A first is that eg_Q preserves the soundness property of the original execution-graph model, namely that it accepts only system-call sequences that are consistent with cfg_Q. A second requirement is that it decays the model's sensitivity to abnormal behavior as little as possible, and therefore preserves the advantage of black-box and gray-box models in that eg_Q should not accept system-call behavior that would not have been observed were it built by training, even though this behavior may be accepted by cfg_Q.

We satisfy the above two requirements by

- creating counterparts of as many nodes and edges in eg_P as possible in eg_Q;
- adding new nodes and edges to eg_Q to accommodate changes between P and Q; and
- performing the above two tasks in such a way that a minimal set of system-call behaviors is accepted by eg_Q.

More specifically, we first copy matched nodes and edges in esg_f to esg_g to the extent possible for all matched function pairs $f \sim g$ (Section 4.1). Next, we handle nodes in cfsg_g that are not matched and create corresponding cross edges (Section 4.2). In the last two steps, we further process the function call nodes to account for the functions they call (Section 4.3) and connect execution subgraphs together to obtain the execution graph eg_Q (Section 4.4).

4.1 Copying nodes and edges when $f \sim g$

The first step, called $\mathsf{copy}()$, in our conversion algorithm is to copy matched portions in esg_f to esg_g, if $f \sim g$. This is an important step as it is able to obtain a large portion of eg_Q, assuming that there is little difference between P and Q, and that the binary difference analysis that precedes our conversion produces common induced subgraphs $\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle$ that are fairly complete for most $f \in P$. Intuitively, for two matched functions f and g, we simply need to copy all nodes and edges in esg_f that are matched and update the names of the nodes (which denote return addresses). However, when a cross edge is copied to esg_g, we need to make sure that there is a full path in cfg_Q that can result in the newly added cross edge (i.e., to make sure that it is supported by a full path).

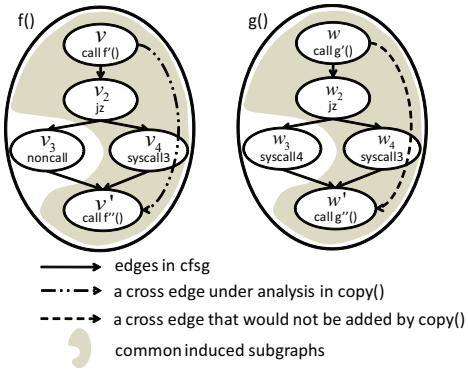

Fig. 1. Cross edge that is not copied

There are two caveats to which we need to pay attention. The first is that a cross edge in esg_f supported by a pruned path containing edges in $\mathsf{cfsgE}_f \setminus \mathsf{isgE}_f$ should

not be copied to esg_g, because something has changed on this pruned path and may render the cross edge not supported in cfg_Q. To improve efficiency, here we restrict our analysis within f and g only and require that all pruned paths (instead of full paths) supporting the cross edge to be copied be included in isg_f and isg_g.

For the example in Figure 1, a cross edge (v, v') is supported by the pruned path $\langle v, v_2, v_3, v' \rangle$ in cfsg_f (which is also a full path). However, there is no pruned path in isg_g that supports the corresponding cross edge in esg_g (so no full path in cfg_Q will support it). The only pruned path $\langle w, w_2, w_4, w' \rangle$ in isg_g does not support this cross edge since this pruned path would unavoidably induce a system call. Thus, the cross edge (v, v') cannot be copied to esg_g.

A second caveat is related to the notion of extended similarity that we introduced in Section 3. Assume $v \sim w$, $v' \sim w'$, and $v'' \sim w''$ (see Figure 2); also assume that $\langle v, v'', v' \rangle \rightsquigarrow (v, v')$. To copy (v, v') to esg_g, we need $\langle w, w'', w' \rangle \rightsquigarrow (w, w')$ and therefore $v'' \overset{\mathsf{ext}}{\approx} w''$ so that any call cycle from v'' can be "replicated" by a call cycle from w'', yielding the same system-call behavior (c.f., Lemma 1).

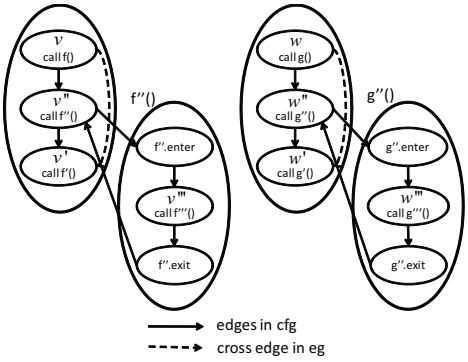

Fig. 2. Extended similarity in $\mathsf{copy}()$

In summary, when a cross edge (w, w') is created in esg_g in this step, all the nodes on the pruned paths supporting this edge are matched, and the nodes along each pruned path not only match but are extended-similar if they are call nodes. We are very strict when copying a cross edge because we do not know which one of the many supporting pruned paths was taken during training of eg_P. In order to avoid possible mistakes in copying a cross edge to esg_g that 1) is not supported by a full path in cfg_Q; or 2) would not have been created had training been done on Q, we have to require that all nodes on all supporting pruned paths be matched and extended-similar. In Figure 3, three cross edges are copied since all the pruned paths that support them are in the common induced subgraph and call nodes are extended-similar.

Algorithm 1 $\mathsf{copy}()$, in Appendix A, performs the operations in this step to copy nodes and cross edges. The following holds for the cross edges it copies to $\mathsf{esg}_g^{\mathsf{cp}}$.

Lemma 2 *Every cross edge added by* $\mathsf{copy}()$ *is strongly supported by a full path in* cfg_Q.

Please refer to Appendix B for an outline of the proof.

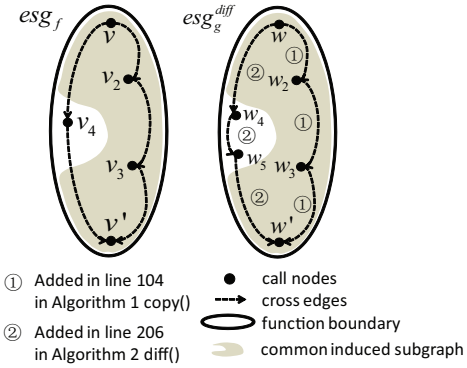

Fig. 3. Converting an execution subgraph

$\mathsf{copy}()$ creates nodes and cross edges by copying them from esg_f. The next step (Section 4.2) shows how we create more nodes and edges for esg_g by statically analyzing the unmatched portion of g.

4.2 The unmatched portion of g

Assuming that f and $g = \pi(f)$ differ by only a small portion, $\mathsf{copy}()$ would have created most of the nodes and cross edges for esg_g. In this step, we analyze the unmatched portion of g to make esg_g more complete. This step is necessary because esg_f does not contain information about the difference between f and g. Intuitively, esg_f and $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$ do not provide enough information for dealing with the unmatched portion of g, and we need to get help from static analysis.

We identify each pruned path in cfsg_g that passes through the unmatched portion of g and then build cross edges between consecutive call nodes on this pruned path until this path is connected to the nodes we created in Algorithm 1 $\mathsf{copy}()$. Three cross edges in Figure 3 are created in this way due to the unmatched nodes w_4 and w_5.

This algorithm, $\mathsf{diff}()$, is detailed in Appendix A. $\mathsf{diff}()$ results in the following property for the cross edges it adds to $\mathsf{esg}_g^{\mathsf{diff}}$; Appendix B gives an outline of the proof.

Lemma 3 *Every cross edge added by* $\mathsf{diff}()$ *is supported by a full path in* cfg_Q.

If there is a cross edge in esg_f that was not copied by $\mathsf{copy}()$ to esg_g, this occurred because a supporting pruned path for this edge was changed (containing unmatched nodes or nodes that are matched but not extended-similar) in g. Whether this pruned path was traversed when P emitted the system-call sequences on which eg_P was trained is, however, unknown. One approach to decide whether to copy the cross edge to esg_g is to exhaustively search (e.g., in $\mathsf{diff}()$) for a full path in cfg_Q that supports it. That is, *any* such path is taken as justification for the cross edge; this approach, therefore, potentially decreases the sensitivity of the model (and also, potentially, false alarms). Another possibility, which reflects the version of the algorithm in Appendix A, is to copy the cross edge only if there is a full supporting path that involves the changed (unmatched) part of g. (This process is encompassed by $\mathsf{refine}()$ in Appendix A, described below in Section 4.3.) In addition to this approach sufficing in our evaluation in Section 5, it better preserves the sensitivity of the model.

4.3 Refining esg_g based on called functions

Many function call nodes have been created in esg_g by $\mathsf{copy}()$ and $\mathsf{diff}()$. Except those extended-similar to their counterparts in cfsgV_f, many of these nodes are created without considering the system-call behavior of the called functions. This is the reason why Lemma 3 claims only that the cross edges created are *supported* but not *strongly supported*. In this step, called $\mathsf{refine}()$, we analyze the system-call behavior of the corresponding called functions and extend the notion of support to strong support for cross edges created so far in $\mathsf{copy}()$ and $\mathsf{diff}()$.

An obvious case in which function call nodes need more processing is when the execution subgraph of the called function has not been created. This happens when the called function g' does not have a match with any function in P. In this case, $\mathsf{esg}_{g'}$ can be obtained by statically analyzing the function itself. For simplicity in this presentation, we reuse $\mathsf{diff}()$ to denote this process in Appendix A, with empty sets for the first three arguments, i.e., $\mathsf{diff}(\emptyset, \langle\emptyset, \emptyset\rangle, \mathsf{cfsg}_{g'})$.

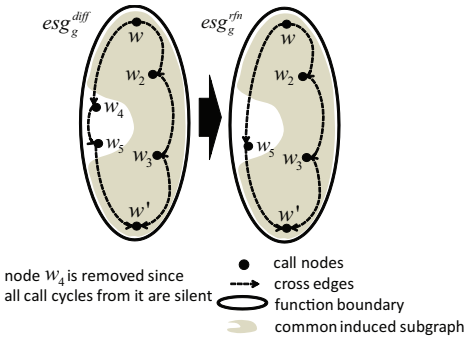

Fig. 4. Function call node removed and cross edges modified

Another scenario in which the function call nodes need more processing is when the called function does not make a system call. Recall that a call node w is created in $\mathsf{copy}()$ and $\mathsf{diff}()$ but we might not have analyzed the called function g' at that time and simply assumed that system calls are made in g' (and therefore these cross edges are *supported* instead of being *strongly supported*). If g' may not make a system call, then we need to either delete w (in the case where g' never makes a system call, shown in Figure 4 where all call cycles from w_4 are silent) or add cross edges from predecessor call nodes of w to successor call nodes of w (in the case where g' may or may not make a system call).

Lemma 4 *After* $\mathsf{refine}()$*, every cross edge in* esg_g *is strongly supported by a full path in* cfg_Q.

Please refer to Appendix B for the proof of Lemma 4.

4.4 Connecting execution subgraphs

At this stage, we create call and return edges to connect all esg_g to form eg_Q. Some of these call edges are created by "copying" the edges from the eg_P, e.g., when the corresponding call node is created in $\mathsf{copy}()$ and is extended-similar to its counterpart in eg_P (case 1 in Figure 5, where $f' \overset{\mathsf{ext}}{\approx} g'$). If a call node w has a match v but is not extended-similar to it, we create an edge (w, w') only for each entry call node w' in the function called by w that matches an entry call node v' for which $(v, v') \in \mathsf{egEcl}_P$ (case 2 in Figure 5, where $f'' \overset{\mathsf{ext}}{\not\approx} g''$), or to all entry call nodes in the called function if there is no such v'. For other call nodes, the call

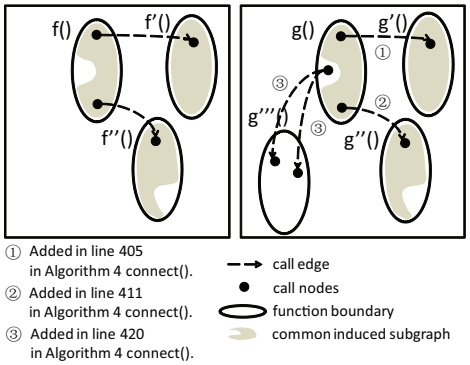

Fig. 5. Using call and return edges to connect execution subgraphs

and return edges cannot be created via copying, and we add call edges between this call node and all the entry call nodes of the called function (case 3 in Figure 5). We create return edges in a similar way.

Appendix A briefly gives an implementation of connect(), and please refer to Appendix B for an outline of the proof of Lemma 5.

Lemma 5 *Every call or return edge added by* connect() *is strongly supported by a full path in* cfg_Q.

Therefore, after running our conversion algorithm, we have a converted execution graph of the new program eg_Q with all the nodes being system call nodes or function call nodes with at least one audible call cycle from each, and all the edges being strongly supported by cfg_Q. Finally, we can state the soundness of our conversion algorithm:

Lemma 6 *The language accepted by* eg_Q *is a subset of the language accepted by* cfg_Q.

This result is trivial given Lemmas 2–5, and consists primarily in arguing that any path q traversed in eg_Q can be "mimicked" by traversing a full path in cfg_Q that travels from each node of q to the next, say from w to w', by following the full path in cfg_Q that strongly supports (w, w').

5 Evaluation

In this section, we evaluate the performance of our conversion procedure. Our conversion program takes in the execution graph of the old binary eg_P, the control flow graph for both binaries cfg_P and cfg_Q, and the output of the binary difference analyzer BinHunt, and outputs the converted execution graph eg_Q of the new binary. We implemented Algorithms 1-4 with approximately 3000 lines of Ocaml code.

We evaluated execution graphs obtained by our conversion algorithm by comparing them to alternatives. Specifically, for each case study, we compared the converted execution graph for the patched program Q with (i) an execution graph for Q obtained by training and (ii) the control flow graph of Q. We performed four case studies.

tar Version 1.14 of tar (P) has an input validation error. Version 1.14-2.3 (Q) differs from P by changing a **do** {} **while**( ) loop into a **while**( ) **do** {} loop (see `http://www.securityfocus.com/bid/25417/info`). This change is identified by BinHunt, but it involves only a function call that does not make any system calls. As such, the system-call behavior of the two programs remains unchanged, and so does the execution graph obtained by our conversion algorithm. (diff() adds a new node and the corresponding cross edges for the function call involved in the change, which are subsequently deleted in refine() because all call cycles from it are silent.)

ncompress In version 4.2.4 of ncompress (P), a missing boundary check allows a specially crafted data stream to underflow a buffer with attacker's data. A check was added in version 4.2.4-15 (Q) to fix this problem (see `http://www.debian.org/security/2006/dsa-1149`). The check introduces a new branch in the program in which an error message is printed when the check fails, causing a new system call

to be invoked. With the same benign inputs for training, the execution graphs for both programs are the same. Our conversion algorithm, however, tries to include this new branch by performing limited static analysis, and consequently expands the execution graph by 3 nodes and 23 edges.

ProFTPD ProFTPD version 1.3.0 (P) interprets long commands from an FTP client as multiple commands, which allows remote attackers to conduct cross-site request forgery (CSRF) attacks and execute arbitrary FTP commands via a long `ftp://` URI that leverages an existing session from the FTP client implementation in a web browser. For the stable distribution (etch) this problem has been fixed in version 1.3.0-19etch2 (Q) by adding input validation checks (see `http://www.debian.org/security/2008/dsa-1689`). Eight additional function calls are introduced in the patched part, most to a logging function for which the execution subgraph can be copied from the old model. The converted execution graph for the patched version thus only slightly increases the execution graph size.

unzip When processing specially crafted ZIP archives, unzip version 5.52 (P) may pass invalid pointers to a C library's `free()` routine, potentially leading to arbitrary code execution (CVE-2008-0888). A patch (version 5.52-1 (Q)) was issued with changes in four functions (see `http://www.debian.org/security/2008/dsa-1522`). Some of the changes involve calling to a new function for which there is no corresponding execution subgraph for the old version. All four changes resulted in static analysis in our conversion algorithm, leading to execution subgraphs constructed mostly or entirely by static analysis. This increased the number of nodes and edges in the resulting execution graph eg_Q more significantly compared to the first three cases.

Experimental results are shown in Table 1 and Table 2. In Table 1, we show the number of nodes and edges in eg_Q that have their counterparts in eg_P and those that do not. More precisely, if $w \in \mathsf{egV}_Q$ and there is some $v \in \mathsf{egV}_P$ such that $v \sim w$, then w is accounted for in the "borrowed" column in Table 1. Similarly, if $(w, w') \in \mathsf{egEcl}_Q \cup \mathsf{egErt}_Q \cup \mathsf{egEcr}_Q$ and there is some $(v, v') \in \mathsf{egEcl}_P \cup \mathsf{egErt}_P \cup \mathsf{egEcr}_P$ such that $(v, v') \sim (w, w')$, then (w, w') is accounted for in the "borrowed" column. Nodes and edges in eg_Q not meeting these conditions are accounted for in the "not borrowed" columns. As this table shows, increased use of static analysis (e.g., in the case of unzip) tends to inflate the execution graph.

	borrowed from eg_P		not borrowed from eg_P	
	# of nodes	# of edges	# of nodes	# of edges
tar	478	1430	0	0
ncompress	151	489	3	23
ProFTPD	775	1850	6	28
unzip	374	1004	50	195

Table 1. Evaluation: nodes and edges in eg_Q

Table 2 compares eg_Q obtained by conversion with one obtained by training. As we can see, eg_Q obtained by training is only marginally smaller than the one obtained by conversion for the first three cases. They differ slightly more in size in the unzip case, due to the more extensive use of static analysis. When the eg_Q as obtained by conversion is substantially larger than eg_P, as in the unzip case, this is an indication that rebuilding eg_Q by training might be prudent.

	Old binary P				New binary Q						
model	eg_P (trained)		cfg_P		eg_Q (converted)			eg_Q (trained)		cfg_Q	
	nodes	edges	nodes	edges	nodes	edges	time (s)	nodes	edges	nodes	edges
tar	**478**	1430	2633	7607	**478**	1430	14.5	**478**	1430	2633	7607
ncompress	**151**	489	577	1318	**154**	512	13.1	**151**	489	578	1322
ProFTPD	**775**	1850	3343	9160	**781**	1878	17.4	**776**	1853	3351	9193
unzip	**374**	1004	491	1464	**424**	1199	41.6	**377**	1017	495	1490

Table 2. Statistics for four case studies. Numbers of nodes for eg_P and eg_Q are highlighted as representatives for size comparison.

Both converted eg_Q and trained eg_Q are smaller than cfg_Q, which, in our experiments, includes cfsg_g for each g reachable from the first function executed in the binary, including library functions. The numbers presented for cfg_Q do *not* include non-call nodes, function call nodes that do not give rise to audible call cycles, `enter` nodes, or `exit` nodes, to enable a fair comparison with eg_Q (since eg_Q does not contain these nodes). Since eg_Q, when trained, is a function of the training inputs, the gap between the sizes of cfg_Q and eg_Q would presumably narrow somewhat by training on a wider variety of inputs (though we did endeavor to train thoroughly, see Appendix C). Absolute sizes aside, however, Table 2 suggests that our conversion algorithm often retains the precision offered by the execution graph from which it builds, no matter how well (or poorly) trained.

An important observation about our converted execution graphs in these case studies is that the language each accepts includes all system-call sequences output by Q when provided the training inputs. We cannot prove that this will always hold with our conversion algorithm, due to limitations on the accuracy of the binary difference analysis tool from which we build [10]. Nevertheless, this empirically provides evidence that this property should often hold in practice.

The conversion time shown in Table 2 for each eg_Q (converted) is in seconds on a 2.8 GHz CPU platform with 1GB memory, and includes only our algorithm time, excluding binary difference analysis and the construction of cfg_Q. (Binary difference analysis with BinHunt overwhelmingly dominated the total conversion time.) As shown in Table 2, as the changes between P and Q increase in size, more time is spent on analyzing cfg_Q and building eg_Q statically. In the cases of ncompress and unzip, the static analysis needs to be applied to the libraries as well.

6 Conclusion

We have presented an algorithm by which an *execution graph*, which is a gray-box system-call-based anomaly detector that uses a model trained from observed system-call behaviors, can be converted from the program for which it was originally trained to a patched version of that program. By using this algorithm, administrators can be spared from setting up a protected and identically configured environment for collecting traces from the patched program. Our algorithm retains desirable properties of execution graphs, including that the system-call sequences accepted by the execution graph are also consistent with the control-flow graph of the program, and that the sequences

accepted tend to capture "normal" behavior as defined by the training sequences. We have demonstrated the effectiveness of our algorithm with four case studies.

As our paper is the first to study adapting anomaly detectors to patches, we believe it introduces an important direction of new research. There are numerous system-call-based anomaly detectors in the literature. Our initial studies suggest that many other such detectors pose challenges to conversion beyond those we have addressed here.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. S. Basu and P. Uppuluri. Proxi-annotated control flow graphs: Deterministic context-sensitive monitoring for intrusion detection. pages 353–362. springer, 2004.
3. E. Buchanan, R. Roemer, H. Schacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.
4. R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Tech. J.*, 9:3–20, 1998.
5. 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.
6. 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.
7. 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.
8. D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graph for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer & Communication Security (CCS 2004)*, 2004.
9. D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
10. D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS 2008)*, 2008.
11. J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
12. J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 2004.
13. R. Gopalakkrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 Symposium on Security and Privacy*, pages 18–31, 2005.
14. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, pages 151–180, 1998.
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. K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the 5th International Workshop on Information Hiding*, October 2002.
18. D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.
19. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
20. Z. Wang, K. Piece, and S. Mcfarling. BMAT – a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:2000, 2000.

A Algorithms

The notation used in the following algorithms follow the convention we stated at the beginning of Section 4: we use f, v and p to denote a function, node and path, respectively, in cfg_P, and we use g, w, and q to denote a function, node and path, respectively, in cfg_Q. We also continue to use $\in$ to denote not only set membership, but a path being in a graph, as well.

Algorithm 1 $\mathsf{copy}()$ picks cross edges from the old function execution subgraph, when we have matches for the two ends of a cross edge and when there is no change that would potentially affect this edge. We copy the edge into the new function execution subgraph (line 104).

Algorithm 1 copy()

Input: esg_f, $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$, cfsg_f, cfsg_g

100: **for all** $(v, v') \in \mathsf{esgE}_f$ **do**
101: **if** $\exists w, w' : v \sim w$ and $v' \sim w'$ **then**
102: $\mathsf{esgV}_g^{\mathsf{cp}} \leftarrow \mathsf{esgV}_g^{\mathsf{cp}} \cup \{w, w'\}$
103: **if** $\forall p \in \mathsf{cfsg}_f, p \overset{\mathsf{s}}{\rightsquigarrow} (v, v') \; \exists q \in \mathsf{isg}_g : \; \forall v'' \in p \; \exists w'' \in q : v'' \overset{\mathsf{ext}}{\approx} w''$ **then**
104: $\mathsf{esgE}_g^{\mathsf{cp}} \leftarrow \mathsf{esgE}_g^{\mathsf{cp}} \cup \{(w, w')\}$

Output: $\mathsf{esg}_g^{\mathsf{cp}}$

In this implementation of Algorithm 1, we examine all pruned paths that strongly support the cross edge to be copied to esg_g (line 103). When the two functions f and g are similar, it is more efficient to examine the differences between f and g to discover the cross edges that should not be copied. When the differences between f and g are small, this equivalent algorithm is more efficient, in our experience.

Algorithm 2 $\mathsf{diff}()$ modifies $\mathsf{esg}_g^{\mathsf{cp}}$ created in $\mathsf{copy}()$. It analyzes each pruned path that passes through the unmatched portion of g, and tries to create a part of execution graph along each such pruned path and connect it to the rest of the execution subgraph.

Algorithm 3 $\mathsf{refine}()$ uses the system call behavior of each called function to determine if any cross edges should be removed and others used in their places. (Analysis in Algorithm 2 does not account for the behavior of called functions when adding edges.)

Finally, Algorithm 4 $\mathsf{connect}()$ tries to copy call edges and return edges from the execution graph of the old program when we have sufficient matching support (line 405

Algorithm 2 $\mathsf{diff}()$

Input: $\mathsf{esg}_g^{\mathsf{cp}}$, $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$, cfsg_g

200: $\mathsf{esg}_g^{\mathsf{diff}} \leftarrow \mathsf{esg}_g^{\mathsf{cp}}$

201: $U \leftarrow \{w \mid w \in \mathsf{cfsgV}_g \ \wedge (w \notin \mathsf{isgV}_g \vee (\exists v : v \sim w \wedge v \overset{\mathsf{ext}}{\not\approx} w))\}$

202: $U' \leftarrow \{w \mid w \in \mathsf{esgV}_g^{\mathsf{cp}} \vee (w \in U \wedge w \text{ is a call node})\}$

203: **for all** $w \in U$ **do**

204: **for all** $q = \langle w_1, \ldots, w_{|q|} \rangle \in \mathsf{cfsg}_g : w \in q \wedge$
$(\forall i \in (1, |q|) : w_i \neq w \Rightarrow w_i \notin U') \ \wedge \{w_1, w_{|q|}\} \subseteq U'$ **do**

205: $\mathsf{esgV}_g^{\mathsf{diff}} \leftarrow \mathsf{esgV}_g^{\mathsf{diff}} \ \cup \{w_i \mid i \in [1, |q|] \ \wedge w_i \text{ is a call node}\}$

206: $\mathsf{esgE}_g^{\mathsf{diff}} \leftarrow \mathsf{esgE}_g^{\mathsf{diff}} \ \cup \{(w_i, w_j) \mid i, j \in [1, |q|] \ \wedge w_i, w_j \text{ are call nodes} \wedge i < j \wedge \forall k \in (i, j) : w_k \text{ is not a call node}\}$

Output: $\mathsf{esg}_g^{\mathsf{diff}}$

Algorithm 3 $\mathsf{refine}()$

Input: $\{\mathsf{esg}_g^{\mathsf{cp}}\}_g, H = \{\mathsf{esg}_g^{\mathsf{diff}}\}_g$, cfg_Q

300: **while** $H \neq \emptyset$ **do**

301: pick one $\mathsf{esg}_g^{\mathsf{diff}}$ in H

302: $\mathsf{esg}_g^{\mathsf{rfn}} \leftarrow \mathsf{esg}_g^{\mathsf{diff}}$

303: **for all** $w \in \mathsf{esgV}_g^{\mathsf{rfn}} : w$ is a function call node $\wedge$
$w \notin \left\{w' \mid w' \in \mathsf{esgV}_g^{\mathsf{cp}} \wedge \exists v' : v' \overset{\mathsf{ext}}{\approx} w'\right\}$ **do**

304: let g' be the function called by w

305: **if** no call cycle from w is audible **then**

306: $\mathsf{esgV}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgV}_g^{\mathsf{rfn}} \setminus \{w\}$

307: **for all** $w', w'' : (w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}} \ \wedge (w, w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**

308: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \cup \{(w', w'')\}$

309: **for all** $w' : (w, w') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**

310: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \setminus \{(w, w')\}$

311: **for all** $w' : (w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**

312: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \setminus \{(w', w)\}$

313: **else if** all call cycles from w are audible **then**

314: **if** $\mathsf{esg}_{g'}^{\mathsf{diff}} \notin H$ **then**

315: $H \leftarrow H \cup \{\mathsf{diff}(\emptyset, \langle \emptyset, \emptyset \rangle, \mathsf{cfsg}_{g'})\}$

316: **else**

317: **for all** $w', w'' : (w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}} \ \wedge (w, w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**

318: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \cup \{(w', w'')\}$

319: **if** $\mathsf{esg}_{g'}^{\mathsf{diff}} \notin H$ **then**

320: $H \leftarrow H \cup \{\mathsf{diff}(\emptyset, \langle \emptyset, \emptyset \rangle, \mathsf{cfsg}_{g'})\}$

321: $H \leftarrow H \setminus \left\{\mathsf{esg}_g^{\mathsf{diff}}\right\}$

Output: $\{\mathsf{esg}_g^{\mathsf{rfn}}\}_g$

and 407). Otherwise, we build call and return edges based on static analysis (lines 411, 413, 416, 418, 420, and 421).

Algorithm 4 connect()

Input: $R = \{\mathsf{esg}_g^{\mathsf{rfn}}\}_g$, egEcl_P, egErt_P

400: **for all** $\mathsf{esg}_g^{\mathsf{rfn}} \in R$ **do**
401: **for all** $w \in \mathsf{esgV}_g^{\mathsf{rfn}}$ **do**
402: let g' be the function to which w calls
403: **if** $\exists v : v \overset{\mathsf{ext}}{\approx} w$ **then**
404: **for all** $v' : (v, v') \in \mathsf{egEcl}_P$ **do**
405: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w')\}$ where $v' \overset{\mathsf{ext}}{\approx} w'$
406: **for all** $v'' : (v'', v) \in \mathsf{egErt}_P$ **do**
407: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w)\}$ where $v'' \overset{\mathsf{ext}}{\approx} w''$
408: **else if** $\exists v : v \sim w \wedge v \overset{\mathsf{ext}}{\not\approx} w$ **then**
409: **for all** $v' : (v, v') \in \mathsf{egEcl}_P$ **do**
410: **if** $\exists w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}} : v' \sim w' \wedge w'$ is an entry call node **then**
411: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w')\}$
412: **else**
413: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w') \mid w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an entry call node$\}$
414: **for all** $v'' : (v'', v) \in \mathsf{egErt}_P$ **do**
415: **if** $\exists w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}} : v'' \sim w'' \wedge w''$ is an exit call node **then**
416: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w)\}$
417: **else**
418: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w) \mid w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an exit call node$\}$
419: **else**
420: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w') \mid w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an entry call node$\}$
421: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w) \mid w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an exit call node$\}$

Output: eg_Q

B Proofs

Proof of Lemma 1. Since $v \overset{\mathsf{ext}}{\approx} w$, by Definition 1, 2, 3, for a call cycle $\langle v, v_2, \ldots, v_n, v\rangle$ in cfg_P, there will be a call cycle $\langle w, w_2, \ldots, w_n, w\rangle$ in cfg_Q such that $v_i \sim w_i : i \in [2, n]$, and if v_i and w_i are system call nodes, they must make the same system call, so these two call cycles result in the same (possibly empty) sequence of system calls. □

Proof of Lemma 2. If (w, w') is added to $\mathsf{esgE}_g^{\mathsf{cp}}$ in line 104, then consider the cross edge $(v, v') \in \mathsf{esgE}_g$ chosen in line 100. Since $(v, v') \in \mathsf{esgE}_g$, there is a full, silent path $p' = \langle v, \ldots, v'\rangle$ in P that was exercised in training. Consider the pruned path p from v to v' obtained by collapsing each call cycle in p' to its function call node. By line 103, there is a corresponding $q \in \mathsf{isg}_g$ on which every node is extended-similar to its corresponding one in p (and hence $p \in \mathsf{isg}_f$, as well). Then, by Lemma 1 there is a full path q' that strongly supports (w, w'). □

Proof of Lemma 3. If an edge (w_i, w_j) is added to $\mathsf{esgE}_g^{\mathsf{diff}}$ at line 206, then w_i and w_j are call nodes with no call node in between them on q. As such, $\langle w_i, \ldots, w_j\rangle$ is a full, silent path that supports (w_i, w_j). □

Proof of Lemma 4. We first argue that any $(w', w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ at the completion of refine() is supported by a full path. First, if (w', w'') was added to $\mathsf{esgE}_g^{\mathsf{cp}}$ in line 104

and then copied forward (lines 200, 302), or if (w', w'') was added to $\mathsf{esgE}_g^{\mathsf{diff}}$ in line 206 and then copied forward (line 302), then (w', w'') is supported by a full path per Lemmas 2 and 3. Now, suppose that (w', w'') was added in line 308 or 318. Then line 305 (respectively, 316) says that some call cycle from w is silent. So, if the cross edges (w', w), (w, w'') were supported by full paths, then the new cross edge (w', w'') is also supported by a full path. It follows by induction, with Lemmas 2–3 providing the base cases, that any cross edges added in lines 308 and 318 are supported by a full path.

We now show that any such edge is strongly supported. Consider any function call node $w \in \mathsf{esgV}_g^{\mathsf{rfn}}$ at the completion of refine. If $w \in \mathsf{esgV}_g^{\mathsf{cp}}$, then it was added in line 102 because it matched some v (line 101) from which an audible call cycle was traversed during training of eg_P. If $v \overset{\mathsf{ext}}{\approx} w$, then by Lemma 1, there is an audible call cycle from w, as well. If $v \overset{\mathsf{ext}}{\not\approx} w$ or $w \notin \mathsf{esgV}_g^{\mathsf{cp}}$, then w satisfied the condition in line 303 and, if there is no audible call cycle from w, was removed in lines 306–312. □

Proof of Lemma 5. Consider an edge added in line 405. Since both v and v' were witnessed during training eg_P, each is a system call node or has some audible call cycle. Because $v \overset{\mathsf{ext}}{\approx} w$ and $v' \overset{\mathsf{ext}}{\approx} w'$, Lemma 1 implies that each of w and w' is a system call node or has some audible call cycle. Moreover, Lemma 1 guarantees that w' is an entry call node since v' is, and so the call edge (w, w') created at line 405 is strongly supported by a full path. By similar reasoning, each return edge added at line 407 is strongly supported by a full path.

In all other cases in which an edge (w, w') is added to egEcl_Q (in line 411, 413, or 420), connect() explicitly checks whether w' is an entry call node for the function g' called by w (line 402), and so there is a full path supporting (w, w'). Similarly, for each edge (w'', w) added to egErt_Q, there is a full path supporting this edge. Since all nodes in each $\mathsf{esgV}_g^{\mathsf{rfn}}$ are either system call nodes or function call nodes from which there is an audible call cycle, these edges are strongly supported. □

C Training

In this appendix we briefly explain how we collected the traces for the four case studies, since training plays an important role in building the execution graphs. For ncompress and unzip, we tried all operation types and options listed in the online manuals. However, for tar and ProFTPD, we did not train as exhaustively as we did for the previous two cases due to the complexity of tar operations and ProFTPD configurations. Nevertheless, for tar and ProFTPD, we did follow guidelines to enhance the repeatability of the training procedure, as described below.

tar Following the manual (see `http://www.gnu.org/software/tar/manual/tar.pdf`), we trained tar for its three most frequently used operations (*create*, *list* and *extract*) that are introduced in Chapter 2 and with all options described in Chapter 3. The directory and files we adopted for applying those operations were the downloaded source of tar-1.14.

ncompress We trained ncompress on its own source directory for version 4.2.4, using all operations and options described in its online manual (see `http://linux.about.com/od/commands/a/blcmdl1_compres.htm`),

ProFTPD We trained ProFTPD configured using the sample configuration file shipped with the source, and with all commands described in the online manual (see `http://linux.about.com/od/commands/l/blcmdl1_ftp.htm`). We chose to transfer files within the ProFTPD-1.3.0 source directory.

unzip Similar to the training on ncompress, we followed the unzip online manual (see `http://linux.about.com/od/commands/l/blcmdl1_unzip.htm`) and trained the program on the .zip package of version 5.52.