

Improving Host-based IDS with Argument Abstraction to Prevent Mimicry Attacks

Sufatrio¹ and Roland H. C. Yap²

¹ Temasek Laboratories, National University of Singapore
5 Sports Drive 2, Singapore 117508, Singapore
tslsufat@nus.edu.sg

² School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543, Singapore
ryap@comp.nus.edu.sg

Abstract. A popular class of host-based Intrusion Detection Systems (IDS) are those based on comparing the system call trace of a process against a set of k -grams. However, the detection mechanism in such IDS can be evaded by cloaking an attack as a mimicry attack. In this paper, we give an algorithm that transforms a detectable attack into a mimicry attack. We demonstrate on a number of examples that using this algorithm, mimicry attacks can be easily constructed on self-based IDS with a set of k -grams and also a more precise graph profile representation. We enhance the IDS by making use of the system call arguments and process credentials. To avoid increasing the false positives, a supplied specification is used to abstract the system call arguments and process credentials. The specification takes into account what objects in the system that can be sensitive to potential attacks, and highlights the occurrence of “dangerous” operations. With this simple extension, we show that the robustness of the IDS is increased. Our preliminary experiments show that on our example programs and attacks, it was no longer possible to construct mimicry attacks. We also demonstrate that the enhanced IDS provides resistance to a variety of common attack strategies.

1 Introduction

In their seminal work [1], Hofmeyr et al. proposed a biologically-inspired host-based IDS which detects anomalies on a running process. This IDS and its later refinements [2–4], which we will call *self-based IDS*, compare the unparameterized system-call trace of a process against the process’ normal profile stored as a set of k -grams, i.e. short sequences of system calls with length k . In the rest of this paper, we will simply use IDS to refer to self-based IDS.

While self-based IDS seem quite reasonable and have been shown to be effective in detecting intrusions, they can be susceptible to *evasion* or *mimicry attacks* [5, 6] which disguise an attack so that it appears “normal” to the IDS. In the first part of this paper, we investigate the susceptibility of self-based IDS to mimicry attacks. Earlier works [5–8] have pointed out the weaknesses of self-based IDS.

In this paper, we focus on exploring practical attack constructions and whether changing parameter(s) of the self-based IDS can help to prevent such attacks. We present a branch-and-bound algorithm for automatically constructing the shortest mimicry attack on self-based IDS and its variants. Our experimental results show that using larger window sizes or disallowing pseudo-edges (defined in Section 3.2) do not prevent attacks; and furthermore, the shortest length mimicry attacks can be constructed without much computation difficulty even for relatively large window sizes.

Our results extend earlier results that mimicry attacks can successfully evade self-based IDS, thus compromising the security of the IDS. In this paper, we seek to find extensions to self-based IDS which can maintain the good self-based IDS properties but make it more difficult to evade detection. Following the terminology used in [9], which classifies IDS as black, gray or white-box detectors, we adopt black and gray-box approaches to enhance IDS. We remark that white-box approaches do complement black and gray-box enhancements such as ours here, but are beyond the scope of this paper.

We propose a simple extension to self-based IDS which incorporates system call arguments and process privileges. We do not use the actual values of arguments and privileges as this could lead to a higher false positive rate from the IDS. Rather we abstract these values by categorizing them into different classes that are defined by a user-supplied¹ *category specification*. The idea here is that an appropriate category specification will take into account the potential security impact of system call operations on system's objects and resources, e.g. files or directories. This combines a slightly more fine-grained gray-box model with a very simple security model. In addition, the security model additionally also allows for immediate rejection/detection of dangerous system calls.

Our experiments show that our extension does increase the strength of a self-based IDS against mimicry attacks. In the sample programs we investigated, mimicry attacks on the enhanced IDS were no longer possible. Our preliminary results indicate that making the profile more fine-grained has little impact on the false positive rate. This is important since we would like to improve the IDS detection capability but without increasing the false positive rate. We believe that the approach in this paper which abstracts arguments and privileges is easy to apply to self-based and similar IDS models while providing more robustness against an intelligent attacker.

The remainder of this paper is organized as follows. Section 2 discusses related work. We examine mimicry attacks and give a branch-and-bound algorithm for constructing mimicry attacks in Section 3. Section 4 describes our IDS enhancement using argument and privilege abstraction. Section 5 gives the results of our experiments on automatically attacking self-based IDS variants and our enhanced IDS. We discuss our empirical results in Section 6, and conclude in Section 7.

¹ This could be one from the system administrator or program developer.

2 Related Work

Mimicry attacks on self-based IDS were introduced in [5,6]. Wagner and Soto [5] use finite state automata (FSA) as a framework for studying and evaluating mimicry attacks. They show that a mimicry attack is possible because additional system calls which behave like no-ops can be inserted into the original attack trace so that the resulting trace is accepted by the automaton of the IDS model. They demonstrate how a mimicry attack can be crafted from the autowux WU-FTPD exploit. Independently, Tan et al. [6] show attack construction on self-based IDS as a process of moving an attack sequence into the IDS detection’s blind region through successive attack modification. The focus in these works was to demonstrate the feasibility of mimicry attacks, but not on a detailed look at automatic attacks.

Recently, Gao et al. [9] performed a study of black-box self-based IDS and also several gray-box IDS. They investigated mimicry attacks with window sizes up to length 6 and showed the existence of mimicry attacks across the methods and window sizes studied. They demonstrated that various forms of IDS are susceptible to attacks but did not go into details of attack generation. Here we give an automatic attack construction algorithm for self-based IDS and similar IDS models, and show empirically that it is computationally easy to generate attacks on self-based IDS for larger window sizes ranging from $k = 5$ to 11.

There are a number of other gray-box enhancements using run-time information which aim to increase the IDS’ robustness. Sekar et al. [10] propose a FSA model built from both system calls and program counter information. Feng et al. [11] also make use of the call stack to extract return addresses. These enhancements have been evaluated in [9] where it is shown that attacks still can be constructed.

The idea of analyzing arguments of operations for detecting behavior deviance appears in a number of works. For example, [12] shows how the use of enriched command-line data can enhance the detection of masqueraders. Our work in this paper is based on an established self-based IDS model, and focuses on system call arguments. Kruegel et al. [13] make use of statistical analysis of system call arguments which can be used to evaluate features of the arguments such as: string length, string character distribution, structural inference and token finder. It is however unclear whether the statistical approach is robust against mimicry attacks.

White-box techniques which incorporate some form of program analysis can complement gray-box techniques. Giffin et al. [14] present a white-box IDS which makes use of static analysis to counter mimicry attacks. They provide some partial results which show how static analysis can make it more difficult for an attacker to manipulate the process and generate a mimicry attack. However, they do not show that the prevention of mimicry attacks. In this paper, we focus on gray and black box techniques which do not require the analysis of source codes or the binaries of the executables.

Finally, we mention that sandboxing techniques also make use of system call argument checking. The `systrace` system [15] uses system call policies to specify

that certain system calls with specific arguments can be allowed or denied. This can be thought of as being a self-based IDS with a window size of one.

3 Constructing Mimicry Attacks

Before explaining our mimicry attack generation algorithm, let us first establish some definitions. A *trace* is a sequence of system calls invoked by a program in its execution. For our purposes, a trace can be viewed simply as a string over some defined alphabet. In this section, we consider self-based IDS model [1–3] where the alphabet for traces are the system call numbers.

We want to distinguish traces generated by a “normal” program execution versus one where the program has been attacked in some fashion. In this paper, we will look at *subtraces*, which are simply substrings of a trace. We also consider *subsequences*, which differ from subtraces as they are a subset of letters from the trace arranged in the original relative trace order, i.e. need not be contiguous in the trace.

The objective of a self-based IDS is to examine subtraces and determine whether they are normal or not. We will call a *basic attack subtrace*, one which is detected by the IDS. A mimicry attack disguises a basic attack subtrace into a *stealthy attack subtrace* which the IDS classifies as being normal.

3.1 Pseudo Subtraces

A weakness of a self-based IDS which makes use of a normal profile represented as a set of k -grams is that it can accept subtraces which actually do not occur in the normal trace(s). For example, consider the following two subtraces of a normal trace:

$$\langle \dots, m_{i-4}, m_{i-3}, m_{i-2}, m_{i-1}, A, B, C, D, E, \dots \\ \dots, B, C, D, E, F, n_{i+1}, n_{i+2}, n_{i+3}, n_{i+4}, \dots \rangle$$

Suppose the window size is 5, and assume that the subtrace $\langle A, B, C, D, E, F \rangle$ never occurs in the normal trace. This subtrace, however, will be accepted as normal by a self-based IDS since the two 5-grams derived are *present* in the normal profile. We call such a subtrace, a *pseudo subtrace* for window size k , since it is not supported by the actual normal trace, yet passes the IDS detection as all its k -grams are present in the normal profile.

A pseudo subtrace can be constructed by finding a common substring of length $k - 1 + l$ with $l \geq 0$ in two separate subtraces of length $m (\geq k + l)$ and $n (\geq k + l)$ respectively, and then joining them to form a new subtrace of length $m + n - k - l + 1$.² We can then concatenate a pseudo subtrace with a normal subtrace or another pseudo subtrace to create a longer pseudo subtrace. A stealthy attack version of a basic attack is simply a pseudo subtrace in which the basic attack subtrace is its subsequence. Figure 1 illustrates such a process which combines subtraces containing attack sequences interspersed with no-ops.

² For attack construction, we set $l = 0$ so as to put the weakest constraint on mimicry attack construction with window size k .

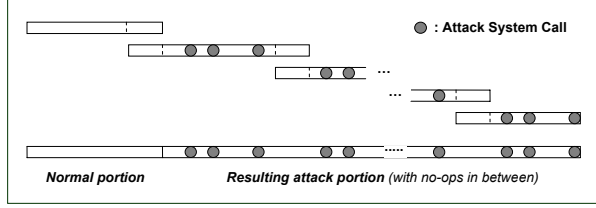


Fig. 1. Mimicry attack construction by composing pseudo subtraces

In this paper, we use the term *pseudo subtrace* to specifically refer to the resulting overall subtrace which is obtained by joining two separate subtraces. The resulting subtrace contains a *foreign sequence* of foreign order type in the terminology of Tan and Maxion [7, 8] with length $k + 1 + l$ as a substring. When $l = 0$ in the joining operation, the foreign sequence is a *minimal foreign sequence*. In the previous example, $\langle A, B, C, D, E, F \rangle$ is a minimal foreign sequence for $k = 5$. The process in Figure 1 constructs a pseudo subtrace for a mimicry attack where minimal foreign sequences of length $k + 1$ may exist along that subtrace, each combining two unconnected subtraces of normal traces together. Here, we emphasise that the core components of mimicry attacks depend on the notions of subtraces and subsequences.³

3.2 The Overlapping Graph Representation

Given a normal trace, we represent a profile using what we call an *overlapping graph*. This is similar to the *De-Bruijn* graph construction used in the “sequencing by hybridization” problem in computational biology [16].

Consider the normal trace of a program N of length n , $\langle N_1, N_2, N_3, \dots, N_n \rangle$, where N_i is the letter representing a system call. Let K be the set of all k -gram subtraces derived from N according to the profile generation rule of a self-based IDS. Given two strings p and q , the function $overlap(p, q)$ gives the maximal length of a suffix of p that matches a prefix of q . The *overlapping graph* G is defined as a directed graph (V, E) where the vertices V are the k -grams in K and the edges E connect two vertices p and q whenever $overlap(p, q) = k - 1$.

We also augment the trace N by adding a suffix consisting of the $k - 1$ occurrences of sentinel symbol, denoted by '\$', signifying the end of the trace. This adds some additional k -grams and simplifies the algorithm.

Figure 2 illustrates the overlapping graph constructed from a normal trace $N : \langle A, B, C, D, E, F, G, A, B, E, F, H \rangle$ with a sliding window of length 3. For simplicity, we have not shown the 3-grams corresponding to $\langle F, H, \$ \rangle$ and $\langle H, \$, \$ \rangle$ which are in G .

³ We remark that not all foreign sequences are pseudo subtraces. Foreign sequences containing system calls not in the k -grams are not considered since they cannot be used to generate mimicry attacks.

There are two kinds of edges in G : *direct edges* and *pseudo edges*. The direct edges are those edges which result from normal substraces. Pseudo edges are those which are not created by two consecutive substrings of length $k - 1$ in the trace. Thus, pseudo edges can be used to generate certain pseudo substraces since it is not in a normal subtrace. In Figure 2, the direct edges are drawn with a single arrow, while the pseudo edges are drawn with a double arrow.

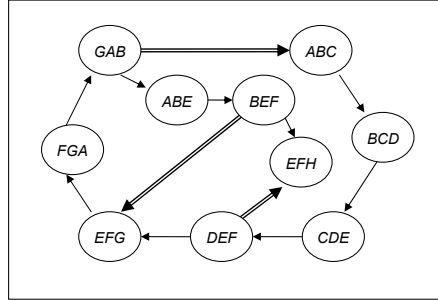


Fig. 2. *Overlapping graph G for $N : \langle A, B, C, D, E, F, G, A, B, E, F, H \rangle$ with $k = 3$*

The graph G can also be viewed as a finite state automata model for recognizing normal traces. A slightly different graph representation is described in [5] where the k -gram database are the state transitions. Their representation however does not distinguish between what in the overlapping graph corresponds to direct and pseudo edges. Since our concern is to address the limitations of self-based IDS, the overlapping graph allows a natural variant where we can evaluate the difference between allowing pseudo edges and removing them.

3.3 Mimicry Attack Construction

Rather than working with the FSA, it is more convenient to directly use the overlapping graph for constructing mimicry attacks. Given an overlapping graph G and a basic attack sequence $A : \langle A_1, A_2, A_3, \dots, A_l \rangle$ which is detectable by the IDS, we want to automatically construct the shortest stealthy attack subtrace $L : \langle L_1, L_2, L_3, \dots, L_m \rangle$ where $m \geq l$ which contains $A_1, A_2, A_3, \dots, A_l$ as a subsequence and where the other system calls in $\{L - A\}$ behave as no-ops with respect to A .

Transforming a basic attack subtrace A into the shortest stealthy subtrace L is equivalent to:

Finding the shortest path P on the overlapping graph G which monotonically visits nodes whose k -gram label begins with the symbol A_i for all $1 \leq i \leq l$.

We augment G with an additional sub-graph, the *occurrence subgraph*. The nodes in the occurrence subgraph, which we will call W , are individual letters for each occurrence of the letter from its k -grams in G . For each node w_i in W , we add an outgoing edge to all nodes in G where the first letter in its k -gram label is the same as the letter for w_i . We call the set of new edges from W to V , the *Occ* set. The resulting graph G' is simply $(V + W, E + Occ)$, which we call the *extended overlapping graph*. Figure 3 shows the extended overlapping graph for the graph in Figure 2.

We illustrate the mimicry attack construction with the following example. Suppose that we want to construct a stealthy subtrace from a basic attack subtrace $A : \langle G, C, D \rangle$ using the extended overlapping graph G' in Figure 3. Note that the subtrace $\langle G, C, D \rangle$ is detected as it is not a 3-gram of the normal trace. Inspecting graph G' , we find the stealthy path: $GAB-ABC-BCD-CDE-DEF$. Thus, the stealthy attack subtrace is the sequence of $\langle \underline{G}, A, B, \underline{C}, \underline{D} \rangle$, with A and B added as no-ops. This example uses the pseudo edge (GAB, ABC) .

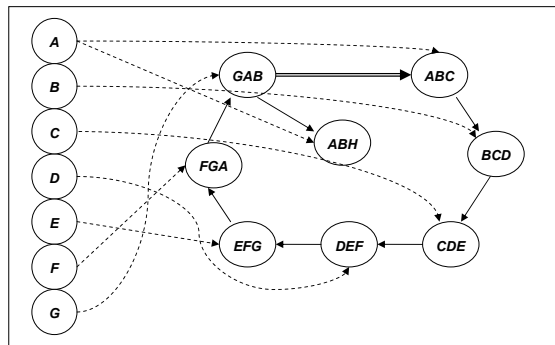


Fig. 3. Extended overlapping graph G' from graph in Figure 2

Our attack construction performs a search to find the shortest mimicry attack. Each node in the search tree corresponds to one letter in the original attack string, A_i . The branches from A_i are the choices of constructing a subtrace starting from the potential k -grams for A_{i+1} pointed to by *Occ*. The process continues until we reach A_l which is the last node in the attack.

In order to make the search more efficient, we employ a *branch-and-bound* strategy to prune the constructed attacks which exceed the best solution found so far. Our implementation uses the *Dijkstra all-pair shortest path* algorithm [17] both to test connectivity between two nodes in G and also to assist in pruning for branch-and-bound search. A sketch of the algorithm is as follows.

Attack Construction Algorithm

Input:

- Sliding window length k
- A normal trace $N : N_1, N_2, N_3, \dots, N_n$
- Basic attack subtrace $A : A_1, A_2, A_3, \dots, A_l$

Output:

- Shortest stealthy subtrace $L : L_1, L_2, L_3, \dots, L_m$
- Or failure, if no solution trace can be found.

1. Perform *Dijkstra all-pairs shortest path* algorithm for all the nodes V .
Between two adjacent nodes, set $distance := 1$.
If two nodes are not connected then $distance := \infty$.
2. Set $Min_distance := \infty$ and $Min_path := \langle \rangle$.
Create a special node v_0 where $\forall i. distance(v_0, v_i) := 0$.
3. Perform *branch-and-bound* search on the *search tree*,
for all $i := 1$ to l choose v_i from $\{v_i | (A_i, v_i) \in Occ\}$:
 - If $distance(v_{i-1}, v_i) = \infty$ then backtrack.
 - Add $distance(v_{i-1}, v_i)$ to $current_cost$.
 - If $current_cost \geq Min_distance$ then backtrack.
 - If complete solution is found then
If $current_cost < Min_distance$ then
 $Min_distance := current_cost$;
 $Min_path := current_path$.
4. Once the search tree is fully explored:
If $Min_distance = \infty$ then return failure;
Else return $L : L_1, L_2, L_3, \dots, L_m$.

In order to use this algorithm in a buffer overflow setting, it needs to be modified to take into account the “border k -gram”. This is further discussed in Section 5.1.

4 IDS Enhancements using Privilege and Argument Abstraction

We now consider a simple gray-box enhancement to an IDS which can either prevent or make mimicry attacks more difficult. To simplify the discussion and evaluation, we will only apply the enhancement to the *baseline* self-based IDS which use system call numbers in the k -grams [1–3].

4.1 Using Arguments and Privileges

In Unix, every process environment contains credentials which are evaluated by the access control mechanism when the process makes a system call. The credentials which determine the current privileges of a process are its effective user-id (euid) and effective group-id (egid). The euid (egid) is either the actual real uid (gid) of the user, or it has been changed by invoking a `setuid` (`setgid`) executable. So, euid and egid are simply a subset of all the user and group-id values defined in a system.

We propose to enhance *k*-gram to include not only the system number but also (abstracted) information about the euid, egid and system call arguments. It is common for attacks to try and exploit programs executing in a privileged mode. The idea is that such attacks can be detected if the corresponding system call subtraces are unprivileged in the normal trace(s). A program which conforms to a good `setuid` programming practice generally drops privileges as soon as possible. Rather than using the actual values, we abstract the euid, egid and system call arguments into categories based on a configuration specification. This is mainly to reduce the false positive rate which can be higher since the space of values is much greater. The abstraction technique also provides flexibility for us to group arguments and privileges together in terms of their importance/sensitivity level.

Formally, we can represent the privilege and argument categorization in the operating system model with the following *mapping* functions:

- Function $EuidCat : U \rightarrow U'$, where: U = the set of euid and $U' \subset \mathbb{N}$ (the set of natural numbers).
- Function $EgidCat : G \rightarrow G'$, where: G = the set of egid and $G' \subset \mathbb{N}$.
- For each $s \in S$ with S = the set of system call numbers, function $ArgCat_s : A_{s,1} \times A_{s,2} \times \dots \times A_{s,max_arg} \rightarrow C_s$, where $A_{s,i}$ for $i \in [1..max_arg]$ = the set of possible entries for i -th argument of the system call s , and $C_s \subset \mathbb{N}$.
Note that it is also possible to omit some arguments, i.e. with wildcards.

As mentioned previously, we can treat the IDS as a FSA model. In the basic self-based IDS, the alphabet was over the system call numbers S , while in the extension, the alphabet is now a tuple $U' \times G' \times S \times C$ where $C = \bigcup_{s \in S} C_s$. Note that while we have focused on Unix, the approach extends to other operating systems.

4.2 A Simple Category Specification Scheme

We now give a simple scheme for defining the abstraction and categories. The category specification is constructed by taking into account the importance or sensitivity level of files/directories in the underlying OS from the security standpoint. The main goal of the specification is to separate operations which have potential security risks from the benign ones.

A fragment of an example specification is as follows:

```

# EUID Abstraction
# Format: <categorized-euid>:<euid1>,<euid2>,...
0:0
1:2000,2001,2003
100:*
# EGID Abstraction
# Format: <categorized-egid>:<egid1>,<egid2>,...
0:0
1:1,2,3,5
100:*
# Argument Abstraction
# Format: <syscall> <arg1> <arg2> <arg3>, ... <cat-value>
open p=/etc/passwd o=0_WRONLY|o=0_RDRW * 1
open p=/etc/shadow o=0_WRONLY|o=0_RDRW * 2
open * * * 18
chmod p=/etc/{passwd,shadow,group,hosts.equiv}|p=/proc/kmem * - 1
# Illegal Transitions Section
# Format: <syscall> <cat-values> [<cat-euid>, <cat-egid>]* ...
open [1..6,8-11,13-15] 0,* *,0
{chmod,fchmod,chown,fchown,lchown,mknod,unlink} 1 0,* *,0

```

The example consists of four sections: euid, egid, argument categorization and illegal transitions. This example is only meant to be illustrative.

Privilege Abstraction The euid and egid section are meant to provide the actual value mapping for $EuidCat : U \rightarrow U'$ and $EgidCat : G \rightarrow G'$. The example specification uses the following syntax for euid and egid:

$$\begin{aligned} \langle u'_i \rangle &: \langle u_{i1} \rangle, \langle u_{i2} \rangle, \dots, \langle u_{in} \rangle; \\ \langle g'_i \rangle &: \langle g_{i1} \rangle, \langle g_{i2} \rangle, \dots, \langle g_{in} \rangle; \end{aligned}$$

where $u'_i \in U'$, $u_{ij} \in U$, $g'_i \in G'$ and $g_{ij} \in G$.

To ensure that $EuidCat$ ($EgidCat$) is a total mapping, a special entry “*” is employed to indicate other euids (egids) so that the mapping satisfies the requirement for a function. As euid=0 and egid=0 signify important privileges in Unix, each of them has a distinguished mapping.

Argument Abstraction The specification is a straightforward one. It maps the system call together with its corresponding arguments (defined in an argument specific fashion, i.e. understands pathnames for `open`) into a number (its category). One point to note that while it is possible to have more complex abstractions, we have found it sufficient to only use a single abstract value to represent multiple arguments. We now briefly discuss some considerations in creating a definition:

- The approach we have used is to focus the specification to a subset of system calls $S' \subset S$ which should be checked in order to prevent attacks aimed

at gaining full control of the system. Our choice for the system call subset S' is based on the work of Bernaschi et al. [18] which classifies Unix system calls according to their *threat level* with respect to system penetration. Here, we consider S' to be the system calls in *Threat-Level 1 Category* of [18], namely: `open`, `chmod`, `fchmod`, `chown`, `fchown`, `lchown`, `rename`, `link`, `unlink`, `symlink`, `mount`, `mknod`, `init_module` and `execve`.

Other system calls in $S - S'$, which have not been defined in the specification are mapped to a unique default value. We do not address the issues raised by the system calls in Threat-Level 2 (can be used for a denial of service) and 3 (can be used for subverting the invoking process) as otherwise we might need a richer IDS model which can also deal with issues such as: memory/storage consumption metering, file access pattern, etc, which are beyond the scope of this paper. One advantage of the system call subset, which is approximately 10% of the total number of system calls, is that it reduces monitoring overheads which is important when the IDS is run on-line.

Bernaschi et al. also groups `setuid/setgid` system call family into the Threat 1 list. However, we take a different approach here in that we capture the effect of the `setuid/setgid` system call family as changes in process credential values –in the form of (euid, egid) pairs– to form part of the state information in our enhanced IDS model.

- Given a system call $s' \in S'$, a simple approach for the choice of abstraction is to ensure that any critical operations on security-sensitive objects are mapped to a value different from a normal one.
- It is convenient, when specifying the abstractions and categories to make use of sequential matching from the start to the end of the definition. In this fashion, more specific mappings can be made first and the most general ones last. This is similar to the ordering in firewall configuration files.
- Pathnames require special treatment and we use a special notation,

$p = \langle \textit{pathname} \rangle$.

Because pathnames in Unix are not unique, they have to be made canonical by turning them into a normalized absolute pathname (see also [15]).

4.3 Disallowing Transitions

It is also useful to specify the transitions that can lead to “bad states”. The idea is to identify those singleton system calls with the corresponding privileges which can be sufficient to compromise the system’s security. An example would be the operation of `chown()` on `/etc/passwd` with root privileges. Thus, the usual way of measuring anomaly signal by means of LFC function as in [2] is not adequate. This can also be used as an enhancement to access control to actually deny such a system call invocation in a program.

Our category specification defines bad transitions as:

$$s' \quad c \quad [u', g']^*$$

where c is the abstracted value for the arguments of system call s' , u' and g' are the abstracted privileges for user and group. Let D_0 be the set of bad transitions.

This specification may be too strict and needs to be adjusted with respect to the normal traces. When the normal profile is extracted from the normal trace dataset, we collect in the set D_N , those transitions from D_0 which match against normal traces. The final adjusted negative transitions are $D = D_0 - D_N$. The IDS detection then concludes that any system call in an execution trace matching an illegal transition $d \in D$ constitutes an intrusion. In addition, we may also prevent the operation itself.

5 Experimental Results

We present the construction of the shortest stealthy attacks on the two variations of self-based IDS and our improved IDS. The three IDS variants are given in Table 1. In Tables 2 to 4, a dash (-) indicates that no stealthy attack could be constructed. We then experiment with our improved IDS against various mimicry attack strategies and investigate its false-positive rate.

The category specification used in these experiments uses the system call subset discussed in Section 4. For the choice of arguments, from the Table 4 in [18], we can see that the dangerous arguments for system calls in S' are mainly files/directories. Garfinkel and Spafford (Appendix B) [19] gives a comprehensive list of security sensitive and important files/directories that one might want to consider monitoring in Unix. In the experiments, we have used a sample generic configuration with several files which are security critical in the Unix/Linux environment: user and group related files (`/etc/passwd`, `/etc/shadow`, `/etc/group`), kernel memory device (`/proc/kmem`), and system configuration files (`/etc/hosts.equiv`). We have omitted for simplicity most of the system configuration files in `/etc` (such as: `/etc/inetd.conf`, `/etc/hosts`, `/etc/cron/*`) and devices files in `/dev`. We have however included entries for various directories commonly found in the Unix/Linux file system hierarchy conforming to the *Filesystem Hierarchy Standard* (<http://www.pathname.com/fhs/pub/fhs-2.3.html>). While one can use a more detailed specification, this is already sufficient to show an increase in IDS robustness.

IDS Model	Remark
IDS-1	Self-based IDS with normal profile stored as a <i>set</i> of k -grams [1, 2]
IDS-2	Self-based IDS with normal profile stored as a <i>graph</i> of k -grams (with only direct edges allowed)
IDS-3	Our improved IDS with normal profile as a <i>set</i> of k -grams

Table 1. IDS models used in mimicry attack construction

5.1 Attack Construction: Baseline vs Improved Self-Based IDS

We extend the attack construction algorithm to also work with our improved IDS model. This is easily done since it just increases the amount of state per node in the graph with the `euid`, `egid` and argument category value.

Our automatic attack construction is implemented in C on a PC with a Pentium 4 processor (1.82 GHz) with 256 MB of RAM running Redhat Linux. We have used also various older versions of the Redhat Linux distribution so as to be able to run the traces corresponding to older versions of programs together with their exploits. The traces are captured in Linux by using the `strace` utility. For simplicity, we have removed system calls which are related to signal events such as `SIGALRM`, `SIGCHLD`, etc. due to their asynchronous nature. In addition, we have purposely set the `euid` and `egid` value of all system call entries in the normal trace to 0. This is to provide the worst-case condition for attacks to occur, i.e. we assume a poorly written `setuid` program.

Remarks on the Exploits Used As our objective is to investigate the practicality of automated attack construction, we experiment with real programs using existing real exploits. Here, we have considered two attack scenarios: (i) *buffer-overflow scenario*: where we can replace the shellcode of a buffer-overflow exploit with a code sequence executing a stealthy attack trace; and (ii) *direct attack*: which might be the result of replacing the program by a trojan which then executes a stealthy trace to fool the IDS.

The following remarks apply to our experiments:

- The three exploits make use of `execve()` system call to spawn a root shell. However, `execve()` is not present in the normal trace. Therefore, we use an alternative strategy to write an entry to the file `“/etc/shadow”`. This actually corresponds to *Attack-strategy* A_2 from our list of strategies shown in Section 5.2. This particular attack strategy is chosen for detailed comparison here as it has been used for mimicry attacks in self-based IDS (e.g. see [6]). We remark that it is perfectly all right to modify the original attack since we assume an intelligent adversary.
- In the buffer-overflow case, there is another constraint that the stealthy attack trace must be introduced at the “attack-introduction point” or “point of seizure”. Hence, we need to manually determine this point and make note of the k system calls before the attack point, which we call a *border k -gram*. Given this, we need to ensure that the concatenation of border k -gram and the stealthy attack trace still passes the IDS. Thus, we need to slightly modify the search algorithm as follows: (i) the border k -gram must be included as an additional input which will then define the associated *border-node* in V ; and (ii) the first-level nodes in V are explored during the search only if they are connected to the border node (and with *path_length* $> k - 1$).

Traceroot2 (Traceroute Exploit) This traceroute exploit is the one previously used in [6]. It is available at: <http://www.packet-stormsecurity.org/>

0011-exploits/traceroot2.c. The exploit attacks LBNL Traceroute v1.4a5 which is included in the Linux Redhat 6.2 distribution.

The original attack sequence is: `setuid(0), setgid(0), execve("/bin/sh")`. This is changed into: `open(), write(), close(), _exit()`. The result of the attack construction on normal traces generated from three Traceroute's sessions (with a total of 2,789 system calls) for sliding-window sizes from $k=5$ to $k=11$ is given in Table 2.

Traceroute Search Result	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	46	48	48	64	64	112	116
IDS-2 (Buffer-Overflow Case)	48	48	64	64	116	116	125
IDS-3 (Buffer-Overflow Case)	—	—	—	—	—	—	—
IDS-1 (Direct-Attack Case)	41	45	45	51	51	54	54
IDS-2 (Direct-Attack Case)	43	45	51	51	54	54	56
IDS-3 (Direct-Attack Case)	—	—	—	—	—	—	—
Average Search Time (User+Sys)	0.170s	0.210s	0.250s	0.300s	0.460s	0.388s	0.340s

Table 2. Attack construction for Traceroute with $k=5-11$ (2,789 Sys-calls in Normal)

Joe Search Result	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	20	30	49	76	79	80	81
IDS-2 (Buffer-Overflow Case)	30	49	76	79	80	81	82
IDS-3 (Buffer-Overflow Case)	—	—	—	—	—	—	—
IDS-1 (Direct-Attack Case)	7	7	7	7	7	7	7
IDS-2 (Direct-Attack Case)	7	7	7	7	7	7	7
IDS-3 (Direct-Attack Case)	—	—	—	—	—	—	—
Average Search Time (User+Sys)	0.258s	0.305s	0.362s	0.432s	0.520s	0.623s	0.778s

Table 3. Attack construction for Joe with $k=5-11$ (9,802 Sys-calls in Normal)

JOE Text Editor Exploit The victim program that we chose is a popular Linux terminal text editor Joe available on <http://sourceforge.net/projects/joe-editor/>. The exploit for Redhat is available at <http://www.uhagr.org/src/kwazy/UHAGr-Joe.p1>, and was run on Redhat 7.3.

Joe is not normally run as a `setuid` program. As a proof of concept, we assume that Joe has been run as root or `setuid` to root. The original attack sequence is:

Wu-Ftpd Search Result	<i>k=5</i>	<i>k=6</i>	<i>k=7</i>	<i>k=8</i>	<i>k=9</i>	<i>k=10</i>	<i>k=11</i>
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	92	182	196	230	256	272	321
IDS-2 (Buffer-Overflow Case)	182	194	212	244	272	303	318
IDS-3 (Buffer-Overflow Case)	—	—	—	—	—	—	—
IDS-1 (Direct-Attack Case)	77	167	181	201	234	257	285
IDS-2 (Direct-Attack Case)	167	179	183	222	257	285	314
IDS-3 (Direct-Attack Case)	—	—	—	—	—	—	—
Average Search Time (User+Sys)	2.036s	2.663s	3.535s	5.056s	4.980s	6.220s	7.811s

Table 4. Attack construction for Wu-Ftpd with $k=5-11$ (11,051 Sys-calls in Normal)

`setuid(0)`, `execve ("/bin/sh")`. Again, we changed it to: `open()`, `write()`, `close()`, `_exit()`.⁴

The result of attack construction on Joe’s normal traces generated from three Joe sessions (with a total of 9,802 system calls) for sliding-window sizes from $k=5$ to $k=11$ is given in Table 3.

Since Joe is an editor, it falls into the class of general purpose programs as opposed to the more privileged processes targeted for monitoring by self-based IDS in [1, 2]. We however include it here to highlight some points on our attack construction results. Note that the search using Attack-strategy A_2 on IDS-3 fails as Joe was not previously used to open `/etc/shadow` in the normal traces.

Autowux WU-FTPD Exploit This is the same exploit previously used in [5]. The `autowux.c` exploits “site exec” vulnerability on the WU-FTPD FTP server. It is available at <http://www.securityfocus.com/bid/1387/exploit/>. We ran the `wu-2.4.2-academ` [BETA-15] `wu-ftp` that comes with Redhat 5.0 distribution on the 2.2.19 kernel.

We use the same attack trace as [5] which is: `setreuid()`, `chroot()`, `chdir()`, `chroot()`, `open()`, `write()`, `close()`, `_exit()`. The result of attack construction on the WU-FTPD normal traces generated from 10 sessions (11,051 system calls) for sliding-window sizes from $k=5$ to $k=11$ is given in Table 4.

Wagner and Soto [5] give a stealthy trace for $k=6$ with 135 stealthy system calls based on their normal profile. Their result, however, is not comparable to ours as the normal traces used are different. In their case, they had collected normal traces for an existing Wu-Ftpd with large numbers of downloads over two days. We have used a small normal profile.

⁴ From the normal traces collected for Joe, we note that there are actually some differences between the normal traces and the exploit trace before the point of seizure due to some `brk()` system calls. This is probably due to increased memory allocation for the buffer overflow attack. However, as reasoned by [5], small differences may be tolerated by the IDS depending on the parameters used in the anomaly signal measurement function of self-based IDS (e.g. Locality Frame Count).

5.2 Behavior of the Improved IDS

Resistance Against Various Attacks Having shown that the improved IDS can better withstand mimicry attacks, we now evaluate the IDS against a number of different attack strategies.

First, we list some important files from the security viewpoint, namely F_1 : `/etc/passwd`, F_2 : `/etc/shadow`, F_3 : `/etc/group`, F_4 : `/proc/kmem` and F_5 : `hosts.equiv`. Next, in Table 5, we list a number of common attack strategies in the Unix/Linux environment on those files above when the system calls are executed with superuser `uid/egid` privilege. While the list is not comprehensive, it suffices to demonstrate improvements in the resistance level of the IDS. We chose the Traceroute program for this experiment. The experiment was done on normal traces described earlier (2,789 system calls) with a sliding-window size set to 5. We found that all the attack strategies listed in Table 5 fail on the tested normal traces even in the direct-attack search scenario. For most of the strategies ($A_6 - A_{61}$), the attacks fail because the needed attack system calls do not appear in the normal traces. In attacks $A_1 - A_5$, given the category specification, the attack searches fail because the normal traces do not contain the particular categories.

ID	Operation (respectively)
$A_1 - A_5$	Open and write an entry into F_1, F_2, F_3, F_4, F_5
$A_6 - A_{10}$	Chmod on F_1, F_2, F_3, F_4, F_5
$A_{11} - A_{15}$	Fchmod on F_1, F_2, F_3, F_4, F_5
$A_{16} - A_{20}$	Chown on F_1, F_2, F_3, F_4, F_5
$A_{21} - A_{25}$	Fchown on F_1, F_2, F_3, F_4, F_5
$A_{26} - A_{30}$	Lchown on F_1, F_2, F_3, F_4, F_5
$A_{31} - A_{35}$	Rename F_1, F_2, F_3, F_4, F_5 into some other file
$A_{36} - A_{40}$	Rename some other file into F_1, F_2, F_3, F_4, F_5
$A_{41} - A_{45}$	Link F_1, F_2, F_3, F_4, F_5 into some other file
$A_{46} - A_{50}$	Link some other file into F_1, F_2, F_3, F_4, F_5
$A_{51} - A_{55}$	Unlink F_1, F_2, F_3, F_4, F_5
$A_{56} - A_{60}$	Mknod F_1, F_2, F_3, F_4, F_5
A_{61}	Execve shell or command

Table 5. Attack strategies to be prevented

False-Positive Rate We give some preliminary results comparing the new IDS in terms of its false-positive rate to the baseline self-based IDS. We chose two programs: `ls` and `traceroute` in Redhat Linux 7.3. For each program, we produced 10 trace sessions and then randomly chose one to be tested against the other 9. The results are shown in Table 6 below. Here we simply measure the number of foreign k -grams. As can be seen, the enhancement does not increase the false positives.

k	Traceroute		ls	
	IDS1	IDS3	IDS1	IDS3
5	0	0	2	2
6	0	0	2	2
7	0	0	2	2
8	0	0	2	2
9	1	1	2	2
10	2	2	2	2
11	3	3	2	2

Table 6. Number of foreign k -grams in Traceroute and ls

6 Discussion

We have shown that the improved IDS model is more resistant to mimicry attacks since the basic attacks in our experiments could not be turned into mimicry attacks. The running times also show that our automated attack construction algorithm is practical and efficient. Execution times for all cases is at most a few seconds on large window sizes. We have the following further observations:

- There can be a considerable difference in length between a stealthy buffer overflow attack compared to the direct attack one for self-based IDS. In some cases, like in Joe, the non-buffer overflow stealthy attack is very short. Here an attack of length seven works for window sizes from $k=5$ to 11.⁵
- The length of the shortest stealthy attack trace varies from program to program. It confirms earlier reports [6, 9] that a larger window tends to require also a longer stealthy attack trace. However, it clearly shows that relying the baseline IDS with certain length of sliding window of, such as six as suggested in [1], is not sufficient. Rather, other improvements are necessary. Our IDS with categorization techniques seems to be able to answer the need to make the self-based IDS more robust. In addition, one can always specify his/her own specification rules in our IDS to suit a particular program in preventing possible attack strategies.
- Our experimental results show that with the given basic attacks, it was not possible to turn them into mimicry attacks on the enhanced IDS although it was possible to do so in the baseline versions of the IDS. Most results that we are aware of for attacking IDS, in particular with mimicry attacks, are usually of the negative variety in that they show potential problems or ways of attacking the IDS. It is significant that our result here is a positive one, since it shows that certain systematic attacks fail to work.

⁵ The actual trojans will usually have longer sequences since there are system calls typically invoked at the beginning of a program related to libraries loading or memory allocation. However, the number does establish the lower-bound of mimicry attacks in the direct-attack setting.

However, we do not guarantee that no attacks are possible since the evaluation is relative with respect to a given basic attack and the normal traces. The question of a security guarantee is in fact an open problem in most IDS models, and we argue that the work here points a way towards more robust evaluation methods.

- We can see that removing pseudo edges for the self-based IDS (the IDS2 model), does not make the IDS significantly stronger against mimicry attacks. In other words, pseudo subtraces can still exist. To understand why, let us consider a normal trace $\langle A, B, C, D, E, A, B, C, M, N \rangle$ with $k = 3$. Given a graph without pseudo edges for this trace, a stealthy trace can still be constructed for a basic attack trace $\langle E, B, D \rangle$. The reason for this is that a common node ABC allows us to create a “crossover path” (i.e. one like $EAB-ABC-BCD$) that makes a stealthy trace possible.
- The false-positive rate experiment is encouraging as it shows that improving the IDS with a more fine-grained detection mechanism does not increase the false-positive rate over the baseline IDS. This means that the IDS is now more accurate for the negative cases as it decreases false-negative rate, but without impacting on the false-positive rate.
- We also can apply the arguments and privileges abstraction technique to other gray-box IDS models, such as the FSA model as in [10]. In this new model, the set of states $\mathcal{Q} = \{q_0, q_\perp\} \cup \{U' \times G' \times P\}$ with $P =$ set of possible program counter values and $\Sigma \in \{S \times C\}$. The transition is thus enhanced using a tuple with the system call number and argument category value.

7 Conclusion

We have presented an efficient algorithm for automated mimicry attack construction on self-based IDS. This is useful for evaluating the robustness of the IDS to attacks. We propose an extension to self-based IDS using privilege and argument abstraction. We argue that this extension is both simple to use and also makes the IDS more robust. Our experimental results show that mimicry attacks which could work in the baseline setting fail in the extended IDS. Hence, the extended IDS is more robust because a more fine grained model which takes into account security aspects of the operations is used. We also have some evidence that the increase in detection accuracy does not lead to more mis-predictions.

An important advantage of our IDS extension is its *simplicity*. Directly using the arguments or process credentials as part of the state will not work well. However, a simple classification which abstracts away irrelevant information and takes into account a security model does work. Furthermore, the simplicity means that it is easy to integrate into various IDS and also can be easily combined with other gray-box techniques to get a significantly more secure IDS.

Acknowledgements

We wish to thank Kymie Tan and the anonymous referees for some helpful comments. We acknowledge the support of the Defence Science and Technology Agency and Temasek Laboratories.

References

1. S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
2. A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, 2000.
3. A. Somayaji. Operating system stability and security through process homeostasis. *Ph.D. Thesis*, University of New Mexico, 2002.
4. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.
5. 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.
6. K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Understanding an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.
7. K. M. C. Tan and R. A. Maxion. Determining the Operational Limits of an Anomaly-Based Intrusion Detector. *IEEE Journal on Selected Areas in Communications, Special Issue on Design and Analysis Techniques for Security Assurance*, 21(1):96–110, 2003.
8. K. M. C. Tan and R. A. 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*, 2002.
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. 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*, 2001.
11. 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*, 2003.
12. R. Maxion. Masquerade detection using enriched command lines. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN-03)*, 2003.
13. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, 2003.
14. J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Network and Distributed System Security Symposium*, 2004.
15. N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
16. P. A. Pevzner. L-tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure and Dynamics*, 7:63–74, 1989.

17. A. V. Aho and J. D. Ullman. *Foundations of Computer Science: C Edition*. W H Freeman & Co, 1995.
18. M. Bernaschi, E. Gabrielli, and L.V. Mancini. REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1):36-61, 2002.
19. S. Garfinkel and G. Spafford. *Practical Unix Security, 2nd Edition*, O'Reilly and Associates, Sebastopol, California, 1996.