svIDS: A Hidden Markov Models Intrusion Detection System

Timothy Sakharov and Oleg Vaskevich

Northeastern University, Boston MA 02115, USA,
`{sakharov.t,vaskevich.o}@husky.neu.edu`

Abstract. In this paper we propose and describe svIDS, an implementation of an intrusion detection system and enforcer based on Hidden Markov Models. The HMM is trained in an unsupervised manner, and then employed by an enforcer daemon to flag potentially malicious system operations. While there is a miscellany of metrics that could be used to characterize a program, svIDS currently models file `open` system calls, and determines that a set of recent operations of a program is suspicious if the file accesses are deemed to have a low likelihood by the trained model. If some threshold of suspicion is met, svIDS's enforcer interdicts the program-under-watch and notifies administrators as necessary.

Keywords: hidden Markov models, intrusion detection system

Table of Contents

1 Introduction

Cybercrime costs the global economy $500 billion dollars a year [1]. Of all the various types of cybercrime, malware and unauthorized intrusion are particularly damaging. For instance, the *Mydoom* worm of 2004 single-handedly caused over $38 billion dollars of damage. Unauthorized access to systems is perhaps even more deleterious. Just two months ago, Kasperksy Labs uncovered a hacking ring that stole over $1 billion from the US through access to ATMs and other banking systems [2]. For this reason, we have chosen to explore malware detection and intrusion detection systems in our *Algorithms and Data* class project, as well as familiarize ourselves with state-of-the-art Bayesian network algorithms and other pertinent machine learning paradigms.

1.1 Project development

Initially, we aimed to apply these algorithms for the purpose of malware detection, but due to the nature of malware with respect to its diversity, we decided to change our focus to intrusion detection by modeling the normal behavior of a specific software program and detecting anomalies.

1.2 Motivation and example use case

The following example is used here and in our implementation as a sample use case and to demonstrate the use of SVIDS.

> Suppose you own a company that provides high-performance computing services to clients, and there is a control panel using a native Linux binary that can only be run off one management node. Each customer has their own user account on this machine, and you design a tool that lets customers query various information, such as the quota or billing amount, as well as change various account settings.
>
> This sensitive data is located on the same machine, but you obviously want to restrict access to it such that only the administrator can make changes (such as increasing the quota) and users can only see their own information. As a result, you place all the data files into a directory accessible only to the root (0) user, and in your tool you set up `setuid(0)` so that the tool runs with administrator privileges.
>
> The problem is that, with each coming week, your company developers keep adding various features to the tool. You're worried that they might introduce a vulnerability into the tool, allowing users to see others' sensitive account information or, worse, modify their own quota and use up all your resources.
>
> Luckily, you have access to SVIDS. You start by creating a `.svids` file with all the expected normal inputs to your program, and SVIDS records the expected observations. Later on, when your program is actually running, SVIDS uses HMMs in order to monitor activity of every

> invocation of the tool. If something looks suspicious—for instance, if the tool is accessing certain directories more often than it should or ones it seldom does (the admin directory with all the secret settings)—then svIDS can interdict and terminate the tool, and notify you and the administrators immediately that something fishy is going on.

While this example is a bit contrived, the main idea is that svIDS is designed to provide an out-of-the-box intrusion detection system and, should it be developed further, become more sound and precise with regards to being able to detect compromises.

The main assumption behind svIDS, shared by other state-of-the-art systems such as kBouncer [3], is that any malicious activity performed must make use of system calls, which can be monitored via `strace` (more on that in subsection 2.4).

While there are many different Linux system calls that can be modeled, svIDS currently models the `open` syscall. As a result, the system is currently designed to detect unlikely directory or file access in a Linux environment. It is particularly helpful when attempting to prevent unauthorized program access to a secure, root-level directory.

For example, if there is software running on a shared system where different user accounts store data that should only be accessible by that user or an administrator, a vulnerability in the software may cause data in a secure directory to be compromised. In order to prevent this, svIDS requires the creation of a `.svids` file with all the expected normal inputs to a given program. svIDS records the expected observations and trains itself on known-good behavior. Later on, when the program is actually running, svIDS operates a real-time "enforcer" that monitors the operations of the program. If something looks suspicious because it's not within the expectations of the model—for instance, some software is accessing directories more often than it should or ones it seldom does (such as the admin directory)—then svIDS can interdict and terminate the program, and then immediately notify the administrators of potential intrusion into the monitored system.

2 Prerequisite concepts

2.1 Markov models

A *Markov model* is a stochastic model used to describe some system with the Markov property that a future state depends only on the present state, as opposed to all the events that preceded it. In particular, it is a system of states with defined probabilities of transitioning from one state to another.

2.2 Hidden Markov models

A *hidden Markov model* has the same underlying idea as a regular Markov model, but with the notion that the state of the system is hidden, and can only be observed through certain *emissions*. These emissions are probabilistic, and thus an HMM is an instance of a simple dynamic Bayesian network.

In a hidden Markov model, we define a single, discrete state variable X_t, where $t \in \mathbb{Z}_0$ indicates the entire state of a system at a certain time t. Note that t need not be represented in a particular unit of time, and can simply be used to indicate a sequence of events.

There are S states and E observations or emissions. Furthermore, X_t has states

$$x_1, \ldots, x_S$$

We define the transmission model

$$T = \Pr(X_t | X_{t-1})$$

as an $S \times S$ matrix, such that

$$T_{i,j} = \Pr(X_t = j, X_{t-1} = i)$$

We also define a sensor or emission model O_t as a set of $E \times E$ matrices, where $O_t \in O$ represents the likelihood of each emission. Finally, we define π_i as the initial probabilities of being in state i. An example of an HMM is shown in Figure 1.

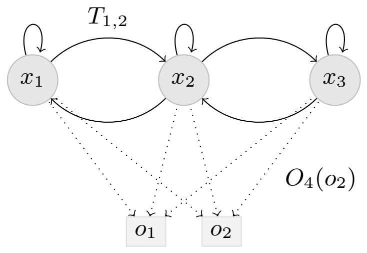

Fig. 1. An HMM with 3 states, each of which emits observations $o_1, o_2 \in O_t$. $T_{i,j}$ is the probability to transition from state x_i to state x_j, and $O_k(x_i)$ is the probability of state x_i emitting o_k.

While HMMs are useful in modeling various systems, they are most powerful when used in tandem with such algorithms as the Baum-Welch and Viterbi algorithms [4].

2.3 Viterbi algorithm

The *Viterbi algorithm* is used to find the most likely sequence of hidden states based on observations, known as the *Viterbi path*. It is defined by the recurrence relation

$$V_{1,k} = \Pr(O_1|x_k) \cdot \pi_{x_k} \text{ where } x_i \in X_1$$
$$V_{t,k} = \max\nolimits_{x \in X_t}(\Pr(O_t|x_t) \cdot T_{x,x_k} \cdot V_{t-1,\text{index}(X_t,x)})$$

The time complexity of this algorithm is $O(tS^2)$. We can determine the Viterbi path by saving pointers to x at each invocation of V.

2.4 `strace` and `ltrace`

`strace` and `ltrace` are two Unix utilities that can be used to observe the system calls and library method calls that a given program makes, respectively. Although it is possible to use these utilities to observe a currently running program, it is typical to invoke them as follows.

```
strace -f <program> <args>
```

The result may look as follows:

```
read(4, "", 8192)                          = 0
close(4)                                   = 0
munmap(0xb770d000, 4096)                   = 0
```

3 Materials and Methods

svIDS is written in a combination of Python 2.7 and Bash.

3.1 Data Representation

We represent our HMM as a JSON object containing a transition matrix, emissions map, and initial probability map—encoded as `transitionmatrix`, `emissions`, and `initprob`, respectively. The transition matrix is the two-dimensional array that represents the probabilities of state transitions. The emissions JSON map, or dictionary, maps a state to its emission. In our case, this is a bijective, one-to-one mapping, as each state has only a single emission with probability of one. This simplification allows for some optimizations in the training algorithm without having too great an impact on its robustness.

This could, however, easily be expanded to account for probabilistic emissions by changing the structure to a one-to-many mapping and modifying the forward-backward procedure in the `update_model.py` program. The emissions represent the subdirectories extracted from the `strace` input, such as 'home' or 'etc'. The initial probabilities object maps each state to the probability of starting in that state. In the case of both `emissions` and `initprob`, the state keys are represented as string forms of integers, such as '2' to indicate State 2, due to the constraints of JSON. This is handled by the scripts to ensure that conversion to and from JSON maintains the representational integrity of the model.

3.2 Code Implementation

The source code for the svIDS project, freely available under Apache License 2.0 on GitHub[1], is divided into three main categories: the trainer, the enforcer, and an example.

Trainer The trainer consists of a Python script and a shell script wrapper. This code parses an svIDS file containing a terminal command and arguments. It then records the system calls of `strace` being run on the given program. These are in turn used to train the Hidden Markov Model by updating the transition matrix, initial probability object, and emissions object as necessary. The program handles the creation and initialization of a new model, revision of existing model entries, and the addition of new emission data if needed. When a given emission is observed repeatedly, the transition matrix for that column (entering into that state) is updated with an increased probability. The initial probabilities work in a similar fashion. Probabilities are smoothed to prevent zero-probability paths through the states.

Enforcer The enforcer uses the trained HMM to analyze a stream of system calls and determine whether the current program execution is suspicious. It does so by finding the probability of the path through the hidden states that is most likely to produce the given emissions (system calls). If the probability of this path is below some predefined threshold, the enforcer will attempt to terminate the program and display a message with relevant information. We have constructed a skeleton for the enforcer program, as the implementation may vary depending on the desired actions taken when a low-probability system call is detected. Since our underlying Hidden Markov Model is encoded in JSON, the components of our system that utilize it, such as the enforcer, are easily adaptable and modifiable.

Example The example portion of the project provides a C program that attempts to exploit a system by accessing directories as a root user. This example can be used to train the svIDS system, or alternatively as an evaluation method to determine whether a trained system is able to detect an intrusion.

Note that, due to time constraints with regards to developing the enforcer, the example code itself is currently incomplete. That being said, it is commented in places where more work is needed, and contriving the rest of the example should be relatively straightforward.

For this reason, we have used the `ls` example also present in this directory, in order to verify that our trainer works as expected.

4 Results

The `out-ls.hmm` file located in our source code in `svids/trainer/` is the result of running the trainer on the `ls` Linux command; note that this is in standard

[1] `https://github.com/ovaskevich/svids`

JSON format. The file contains the subdirectories that the trainer encountered (encoded in the emissions matrix), the initial probabilities for the states, and the transition matrix. The system call being trained in this case is the `open` syscall, so the emitted text in the arguments of the call is that of the directory structure. The advantage of a system like svIDS is that there is no need to provide complex definitions for possibly intrusive software behavior. Instead, even a simple Linux command can be used to provide a baseline, although binaries closer to those used by a particular program will increase accuracy. As long as the trainer is provided with ample data that represents *benign* program execution, it will be able to determine whether some newly encountered system call seems to be intrusive.

5 Conclusion and Possibilities for Future Work

Overall, the Hidden Markov Model behind svIDS provides an expandable and robust solution for intrusion detection in a Linux environment. Although the enforcer is not yet fully implemented, and we currently only model one type of system call, it can easily be written for a specific application of svIDS. The nature of this project was a proof-of-concept: we examined the viability of an HMM for detecting suspicious software behavior. The unsupervised training is perhaps the greatest strength of this approach; the signatures of various intrusion attempts need not to be known. Even in the simple case of using the `ls` command to train the model, it was still able to meaningfully determine a number of benign directories and subdirectories. If given adversarial code, the model would be able to determine that its emissions have a low probability, and therefore flag them as suspicious.

5.1 Future work

svIDS also has plenty of room for future work. Outside the implementation of the enforcer, the system could incorporate some notion of structure for the system calls themselves. Currently, the model only looks at individual system calls from the `strace` output, one at a time, and implements the model for a subset of these. Future work could interpret system calls differently based on their sequence in the input as a *digram*; for instance, a call to `read` followed by a call to `write` would affect the HMM differently from a `read` followed by an `execve`. Another possible expansion would be to incorporate state merging to allow for more nuanced state structure [4].

Bibliography

[1] M. Egan, “Report: Cyber Crime Costs Global Economy Up to $500B a Year,” *Fox Business*, 2013.

[2] F. News, “Hacking ring has stolen up to $1 billion from US, European banks, report says,” *Fox News*, 2015.

[3] V. Pappas, “kBouncer: Efficient and Transparent ROP Mitigation,” 2012.

[4] A. Stolcke and S. Omohundro, “Hidden Markov model induction by Bayesian model merging,” *Advances in neural information processing systems*, pp. 11–11, 1993.