Towards Creating Believable Decoy Project Folders for Detecting Data Theft

Stefan Thaler[1(✉)], Jerry den Hartog[1], and Milan Petkovic[1,2]

[1] Technical University of Eindhoven,
Den Dolech 12, 5600 MB Eindhoven, Netherlands
{s.m.thaler,j.d.hartog}@tue.nl
[2] Philips Research Laboratories, High Tech Campus 34, Eindhoven, Netherlands
milan.petkovic@philips.com

Abstract. Digital data theft is difficult to detect and typically it also takes a long time to discover that data has been stolen. This paper introduces a data-driven approach based on Markov chains to create believable decoy project folders which can assist in detecting potentially ongoing attacks. This can be done by deploying these intrinsically valueless folders between real project folders and by monitoring interactions with them. We present our approach and results from a user study demonstrating the believability of the generated decoy folders.

Keywords: Data theft detection · Data theft · Intrusion detection · Decoy · Honey pot · Trap-based defense · Deception

1 Introduction

Digital data theft is becoming a huge problem in our ever more digitalized society. The total number of data breach incidents as well as the damage caused are rising an alarming rate. One strategy to detect ongoing digital attacks is baiting data thieves with digital decoys. These decoys are valueless, therefore any interaction with them is suspicious and will alert the responsible security offer.

One type of decoys is decoy documents. A decoy document is a file which contains seemingly sensitive information. These documents are usually stored with other sensitive documents and closely monitored. Interactions with them are reported and stored. However, while decoy documents are useful tools to obtain intelligence about an ongoing attack, most of a company's intellectual property comprises multiple files and folders, grouped together in a project folder. Yet, deceptive approaches as detection strategy are hardly used in this setting, since manually creating believable project decoys is a cumbersome, labor intensive task.

In this work we present a data-driven approach for dynamically creating new project folders from a set of existing folders based on multiple Markov models using maximum-likelihood parameter estimation. After the decoy model is learned, it can be used to sample believable project folders which resemble

S. Ranise and V. Swarup (Eds.): DBSec 2016, LNCS 9766, pp. 161–169, 2016.
DOI: 10.1007/978-3-319-41483-6_12

the original folders and can be used as deceptive bait. This deceptive bait can be placed between "real" projects. Assuming the decoys are properly monitored, they can aid in the detection of malicious activity around these real project files, since any interaction with a decoy is per definition suspicious. We have implemented a prototype and conducted a user study to evaluate whether the generated project decoys are perceived as being realistic.

We begin this paper with a brief review on related approaches in Sect. 2 to highlight gaps and motivate our approach. In Sect. 3 we describe our approach. We start by giving a high level intuition of our approach which is followed by a formal definition of our decoy model. In Sect. 4, we provide empirical evidence for the believability of the generated decoys. In Sect. 5 we discuss the results and limitations of our work. We conclude this paper by highlighting possible future improvements.

2 Related Work

Deception has been an element of warfare since a long time. In the context of IT security, Clifford Stoll reported one of the first uses of deception [9] to catch a hacker. Lance Spitzner coined the term honeypot and honeytoken [8], which are descriptive synonyms for computing resources that are created to deceive an attacker. Bringer et al.'s survey provides an overview over recent advances in the field [3].

In this paper we introduce a data-driven approach to create believable project folders that can be deployed to detect data theft attempts. Previous approaches focused on creating and placing single decoy documents([2,10,11,13]) or file systems [6]. Conceptually we also create decoys to detect data theft. However, instead of creating single documents we focus on creating project folders with a believable structure, file names and properties.

3 Decoy Project Folder Generation

On a high level our approach can be divided into two phases, a *learning phase* and a *sampling phase*. Both phases are subdivided into two steps each.

The first step of the *learning phase* is to normalize the training data. In this step we replace properties of the source files and folders with placeholders. Examples of such properties are occurrences of the project name or access rights. We use this normalized training data to learn Markov chains representing the folders and property distributions, which is the second step of the learning phase. The decoy model is defined in Sect. 3.2.

In the *sampling phase* we use the previously learned model to generate decoy project folders. The first step of this phase is to generate the decoy directory structure. The second step is to instantiate the decoy project folder by assigning attributes to the nodes of the directory structure and replacing property placeholders with instantiation values.

3.1 Preliminaries and Assumptions

We assume the following concepts used in the paper are known: we will use first-order, finite-state, discrete-time, *Markov chains* (simply called Markov chains below) to model the co-occurrence of files and folders within one directory. We use the *maximizing likelihood estimator* [5] to learn the Markov chain parameters from observed sequences of files and folders.

In order to maximize the information of our learned decoy model, we will use a heuristic based on the *mutual information* [7] of files occurring together.

We will use a (finite) *forest* [4], i.e. a set of trees, to model our training file systems.

Learning requires multiple samples. To be able to effectively learn the content of folders we assume that two folders with the same name have similar contents.

Assumption 1. *If two folders within a context have the same name, they have similar contents.*

3.2 Definition of the Decoy Model

In this section we formally define our decoy model in a bottom-up manner.

To build the model we will use a training file system that contains a number of projects which structure objects such as files and directories in the form of a tree and assigns properties to these objects.

Definition 1. *We assume a fixed set of objects O, a set of attributes $\mathcal{A}$ and a (training) file system TT over these objects is given. Here an* attribute *$a \in \mathcal{A}$ is a function from objects to a domain Dom_a and file system TT is a forest over objects.*

Intuitively, Definition 1 states that we use a set of project folders as input for our decoy model. These project folders are organized within a directory tree, which is common in Windows- and Unix-based operating systems. An object represents a file or a folder within a project directory, and each file or folder has certain attributes such as a name, permissions or creation- and modification dates.

Some attributes are dependent on the project, for example filenames containing the project title or creation time of files that will be within the running time of the project. Therefore, to learn a common structure from different projects we 'normalize' the project to make them consistent. For example, we replace all occurrences of the project name by a 'placeholder'. We then refer to the normalized objects as nodes, and the normalized TT as NodeTree. For simplicity a file is represented as a node without children.

Definition 2. *We assume a fixed set $\mathcal{N}$, called Nodes, a node tree NT over nodes and normalization function $\mu : O \to \mathcal{N}$, mapping objects to nodes, are given.*

The only attribute that we assume is always there is '*name*'. The other attributes may depend on the operating system, therefore we do not further specify them here. Instead, we simply assume they are available.

We aim to learn the content of folders. Assumption 1 states that similar names convey similar contents. Therefore, we group nodes (i.e. normalized files and folders) by their name. Next, we look at the content i.e. the children of these nodes.

Definition 3. *A* ***NameGroup*** $NG(d)$ *for a name* $d \in Dom_{name}$ *is the set nodes that have name* d*. A* ***NameGroupChildren*** $NGC(d)$ *for a name* $d \in Dom_{name}$ *is a set of sets, capturing the content of these folders.*

$$NG(d) = \{n \in N \mid name(n) = d\}$$

$$NGC(d) = \{ch(d') \cup \{s_s, s_e\} \mid d' \in NG(name(d))\}$$

Here, s_s *represents an artificial node 'start state', i.e. the start of a directory and* s_e *represents the end of a directory.*

From a NameGroupChildren we can find the frequencies of occurrence of nodes. However, simply taking their frequency may not be sufficient; some nodes will naturally occur together or will exclude each other. As such combination might allow an attacker to easily spot a fake project folders we need to take such dependencies into account. Thus we need to estimate the probability of certain nodes co-occurring in a folder. Yet, learning the complete joint probability of all nodes with any accuracy is not very realistic because the number of samples needed grows exponential with the number of (possible) children nodes of a node.

As a compromise we use Markov chains to model the probabilities. This allows taking into account pairwise dependencies between some nodes. In order to train our Markov chains we will treat the children of the nodes of a NameGroup as encountered observations. In this construction the order of the nodes matters; dependencies are only taking into account between the current and next node. After visiting an intermediate node this information is lost. As such nodes with a relevant dependency on each other should occur next to each other to ensure the dependency is expressed in the model.

Definition 4. *To help optimize the amount of information about dependencies that we can capture in our model we use a simple, greedy heuristic 'sorted' which starts with the start state* s_s *and then continuously selects follow-up nodes with maximum mutual information* $max(I(n_i, n_{i+1}))$*, where* n_{i+1} *has not been selected before. In case of a tie we prefer nodes that occur together often, i.e.* $\mathcal{P}(n_{i+1}|n_i)$ *is higher. 'sorted' always ends with the end state* s_e*.*

Next, we want to learn a Markov chain with states S and a transition matrix P for a name group NG. That is, we want to learn a Markov chain that represents the distribution of files and folders of all directories with the same name (see Assumption 1). To do so, we treat the sets of nodes $c \in NGC(d)$ as sequence of 'observed events'.

Definition 5. *Let d be a name $d \in Dom_{name}$ and $NGC(d)$ the name group children of d. Then, we define the* ***states*** *S of the Markov Chain MC for a NGC with:*

$$S = (\bigcup_{c \in NGC(d)} c)$$

Furthermore, we will refer to a Markov chain MC that for the NameGroupChildren of a node $NGC(d)$ as MC_d. Next, we estimate transition probabilities of MC_d.

Definition 6. *Let us assume that each $c \in NGC(d)$ has been sorted by the heuristic which was introduced in Definition 4. Then the transition probabilities P for two successive nodes n_i,n_{i+1} of a Markov chain MC_d are estimated using a Maximum Likelihood Estimators, using each $c \in NGC(d)$ as an observed sequence of events.*

So far we have modeled single directories. A tree model captures a whole directory tree.

Definition 7. *The* ***tree model*** *TM is a function that maps a name $d \in Dom_{name}$ to a corresponding Markov chain MC_d.*

$$TM : d \rightarrow MC_d$$

In order to capture the probabilities of the node attributes, we learn the probability that certain nodes have. Here Assumption 1 is used to cluster the group of properties.

Definition 8. *The* ***attribute model*** *captures the frequency of occurrence of an attribute value within a name group. It is a function from $\mathcal{N}$ to probability measures $\mathcal{P}$ over the domain of a. $\mathcal{P}$ is defined as:*

$$\mathcal{P}_{a,NG(n)}(x) = \frac{\#\{n' \in NG(n) \mid a(n') = x\}}{\#NG(n)}$$

and the attribute model AM as:

$$AM(n, a) = \mathcal{P}_{a,NG(n)}$$

Finally, using the previous definitions we define our decoy model as follows:

Definition 9. *The* ***decoy model*** *DM comprises the tree model TM and attribute model AM. Intuitively, the tree model captures the learned, normalized directory structure, whereas the attribute model contains the learned, normalized file- and folder attributes.*

3.3 Sampling Decoys

A learned decoy model can be used to sample decoy project folders. First, the user chooses instantiation values for the placeholders, e.g. a project name for the decoy project. Then, the directory structure is created by a recursive random walk over the tree model. This recursive random walk starts with the Markov chain MC_p, where p is the placeholder for the project name. This random walk generates a set of nodes, which represents the content of this folder. The recursion step performs random walks over the Markov chains for each of these nodes.

Thereafter, we instantiate the generated directory structure, which outputs the decoy project folder. We randomly pick the attributes from the attribute model AM for each node of the directory structure. Finally, we replace all attribute placeholders with the chosen instantiation values and write the instantiated nodes as files or folders to the disk.

4 Validation

Decoys have several desirable properties ([2,12]), but the most important one is believability, i.e. capable of eliciting belief or trust [2]. Therefore, we have conducted a user study to validate whether human judges are able to distinguish between real projects and decoys. To conduct the experiments, we have implemented a prototype of our approach in Python 2.7.

Survey setup. In this study, we presented 30 software project folders to 27 human judges. Half of the presented software project folders were generated using our prototype, the other half were real project directories. The study participants were asked to decided which ones were real and which ones were decoys. The software project folders were shared on-line and the participants could click around freely within the projects.

We learned three different decoy models, one for Java projects, one for Python projects and one for Ruby on Rails projects. We chose three different project types to identify differences in the trained model and the effect of the believability of the decoy. In total we used 25 open source projects for each project type to learn the decoy models. The number 25 was determined using trial and error and based on believability and variability of the generated decoy projects. The "real" projects that we use for training were different from the "real" ones that we presented to the survey participants. We obtained the projects by searching GitHub for certain terms, e.g. "Java and Security" and chose popular projects to related topics. Since evaluating 30 projects would take a long time, the study was split into three parts, one for each project type. Each participant was free to chose which of the parts and the number of parts they wanted to complete. Each partial surveys contained a section with demographic questions.

We recorded the answers to the surveys as well as the clicks of the participants within the folders. We neither counted survey answers where we did not have any corresponding clicks nor clicks on folders that could not be correlated to

participants of the study. The data was collected within the period from February 1st to February 8th in 2016.

In total, we had 27 unique participants and we have collected 23 complete surveys for Java, 19 for Python and 15 for Rails projects resulting in 570 answers. We rejected 27 answers where the monitoring showed that the corresponding project was not looked at by this user. Thus in total we collected 543 valid judgments.

The majority of our study participants (81.5 %) was highly educated, i.e. had a college or post graduate degree and also most of them (70.37 %) rated their own software development skills as average or better. Most of the participants (78.3 % / 68.3 %) who responded to the Java and Python part of survey stated that they have at least some knowledge of the Java programming language / Python programming language respectively, whereas only 20 percent of the participants had some knowledge of the Rails framework. On average participants rated their Java, Python and rails framework knowledge respectively as 2.87, 2.43 and 1.40 on a scale from 1 to 5 with 5 being the highest.

To avoid participants' bias towards project names, we replaced the project names that occurred in all files and folders of a project with a randomly numbered placeholder of the form Pxx, where xx was a randomly chosen number between 01 and 20. The study participants were informed about this measure. Apart from that, the participants knew that they were presented partially generated and partially real folder listings. To counter any bias originating from this knowledge, we phrased the evaluation question in a non-suggestive way, i.e.: "*The number of real/decoy projects does not necessarily have to be balanced. There may be 0 real projects, there may also be up to 10 real projects.*".

Survey results. In total, of the 543 valid answers 271 were on real projects and 272 were on decoy projects. 143 of the answers on real projects labeled them correctly as real and 128 declared them as decoys (52.7 %). 135 of the answers on decoys labeled them incorrectly as real and 137 were correctly identified as decoys (50.37 %). The average accuracy per project type was 52.19 % correct answers for Java projects, 53.63 % for Python projects and 46.32 % for Rails projects. In total 51.20 % of the 543 answers were correct.

The minimum recorded accuracy of the individual judges was 30 %, and about a quarter of the participant's accuracies were below 50 % total accuracy while the highest accuracy was 90 %. The total average individual performance was 53.3 %, which is slightly better than random guessing.

5 Discussion and Limitations

We have presented a data driven approach to create decoy project folder. These decoys may aid detecting potential data theft attacks by placing them closely monitored between real projects. A perfect decoy would be indistinguishable from the object it tries to mimic, therefore ideally the performance of the judges should be close to random guesses. In our believability study, the participants

were able to identify real projects and decoys in 51.57 % of the answers, which suggests that they had difficulty distinguishing generated projects from real ones.

We have empirically shown that the generated decoys are perceived as believable, however, we did so in a limited way. We did not take into account some of the external factors that can influence the believability of the decoy, for example the context where a decoy is deployed, the choice of source projects that are used to train the decoy model as well as the choice of projects that decoys will be compared to during the study. In order to more properly interpret the results of our study, further investigations on a well-founded baseline are required. Finally, we have to validate whether the decoys can be effectively used to detect data-thefts.

Our proposed approach has some limitations. First of all, currently we need to specify instantiation parameters such as the name of the project decoy manually. We assume that these parameters contribute to the effectiveness of our approach, thus ideally they are also learned from the source projects as well as the target context. Next, our approach will not work if an attacker already knows which project they want to steal, because then they do not need to browse other projects to determine their value. Also, in other contexts simply refusing access to the data works better than deceiving an attacker [1]. Finally, potentially sensitive data could be leaked using our approach.

6 Conclusion and Future work

In this paper we presented a data driven approach based on first-order Markov chains that can be used to automate the generation of decoy project folders within a specific, topical context. We have implemented a prototype of this approach, generated decoy project folders and validated their perceived believability via a user study with 27 participants and 543 evaluated projects. The user study showed that the participants could only correctly identify 51.2 percent of the projects, which is only slightly better than random guessing.

To address the previously mentioned limitations and thereby improve our approach, we plan to include methods for learning and generating the patterns of existing file names so that we can create new ones. Furthermore, we believe that hybrid methods which are based on templates as well as on data could further improve the believability of the generated project folder structure. Additionally, we are investigating on methods to learn instantiation parameters such as the project name from existing sources.

While our approach has certain limitations, we envision decoy project folders being deployed as cheap, supportive measure to detect ongoing data thefts.

Acknowledgements. This work has been partially funded by the Dutch national program COMMIT under the THeCS project.

References

1. Biskup, J.: For unknown secrecies refusal is better than lying. In: Atluri, V., Hale, H. (eds.) Research Advances in Database and Information Systems Security. IFIP, vol. 43, pp. 127–141. Springer, New York (2000)
2. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Chen, Y., Dimitriou, T.D., Zhou, J. (eds.) SecureComm 2009. LNICST, vol. 19, pp. 51–70. Springer, Heidelberg (2009)
3. Bringer, M.L., Chelmecki, C.A., Fujinoki, H.: A survey: recent advances and future trends in honeypot research. Int. J. **4**, 63–75 (2012)
4. Diestel, R.: Graph Theory. GTM, vol. 173, 4th edn. Springer, Heidelberg (2010). ISBN 978-3-642-14278-9
5. Kemeny, J.G., Snell, J.L.: Finite Markov Chains, vol. 356. van Nostrand, Princeton (1960)
6. Rowe, N.C.: Automatic detection of fake file systems (2005)
7. Shannon, C.E., Weaver, W.: The Mathematical Theory of Communication. University of Illinois Press, Urbana (1949)
8. Spitzner, L.: Honeypots: catching the insider threat. In: Proceedings of the 19th Annual Computer Security Applications Conference 2003, pp. 170–179. IEEE (2003)
9. Stoll, C.P.: The cuckoos egg: tracing a spy through the maze of computer espionage (1989)
10. Voris, J., Boggs, N., Stolfo, S.J.: Lost in translation: improving decoy documents via automated translation. In: 2012 IEEE Symposium on Security and Privacy Workshops (SPW), pp. 129–133. IEEE (2012)
11. Whitham, B.: Canary files: generating fake files to detect critical data loss from complex computer networks. In: The Second International Conference on Cyber Security, Cyber Peacefare and Digital Forensic (CyberSec2013), pp. 170–179. The Society of Digital Information and Wireless Communication (2013)
12. Whitham, B.: Design requirements for generating deceptive content to protect document repositories (2014)
13. Yuill, J., Zappe, M., Denning, D., Feer, F.: Honeyfiles: deceptive files for intrusion detection. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004, pp. 116–122. IEEE (2004). http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1437806, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA484922