Software-based Decoy System for Insider Threats *

Younghee Park
Department of Computer Science
Columbia University, New York, NY 10027
younghee@cs.columbia.edu

Salvatore J. Stolfo
Department of Computer Science
Columbia University, New York, NY 10027
sal@cs.columbia.edu

ABSTRACT

Decoy technology and the use of deception are useful in securing critical computing systems by confounding and confusing adversaries with fake information. Deception leverages uncertainty forcing adversaries to expend considerable effort to differentiate realistic useful information from purposely planted false information. In this paper, we propose software-based decoy system that aims to deceive insiders, to detect the exfiltration of proprietary source code. The proposed system generates believable Java source code that appear to an adversary to be entirely valuable proprietary software. Bogus software is generated iteratively using code obfuscation techniques to transform original software using various transformation methods. Beacons are also injected into bogus software to detect the exfiltration and to make an alert if the decoy software is touched, compiled or executed. Based on similarity measurement, the experimental results demonstrate that the generated bogus software is different from the original software while maintaining similar complexity to confuse an adversary as to which is real and which is not.

Keywords: decoy, insider attacks, code obfuscation

1. INTRODUCTION

Information theft by insiders, who exfiltrate sensitive information and intellectual property using legitimate credentials, has been a serious problem for decades. Software is one of the most valuable assets for many organizations and enterprises, and is also the most lucrative target for insiders. In 2010, according to the FBI, a former employee of Goldman Sachs, a computer programmer, was accused of stealing trading software by uploading to a server in Germany a program implementing its proprietary trading platform for equity products, and the use of the software made the thief in excess of 300 million dollars in illicit profits in one year. Clearly, it is crucial to create an environment in which the most valuable proprietary software is isolated and protected from such theft.

*This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under the ADAMS (Anomaly Detection at Multiple Scales) Program with grant award number W911NF-11-1-0140.

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

Much previous work has focused on the detection of insider attacks that exfiltrate important and sensitive information, rather than proprietary software. To detect the exfiltration, deception-based defense methods (i.e. decoys) are important mechanisms in the protection of systems, networks, and information. In addition, document-based decoy system has gained lots of interests for the detection of insider threats [2, 1]. This decoy system is presented to confuse and deceive adversaries based on fake information [5, 6, 16].

While previous research has focused on detecting *information* leaks, this paper proposes a software-based decoy system to deceive attackers in order to detect software exfiltration and to track unauthorized uses. The basic concept of software-based decoys is similar to the document-based decoy to detect insider attacks. However, the software-based decoy system prevents insiders that illegally obtain proprietary software through deceptive mechanisms [4, 19, 13]. The objective of this paper is to isolate the proprietary source code from such theft using fake source code as decoys. By using software decoys, the proposed system identifies or detects the exfiltration of proprietary software in enterprises or government organizations from insiders. For the design of software-based decoy system, this paper addresses the two research problems: (1) How to generate fake (bogus) software? (2) How to detect software exfiltration and unauthorized use?

First, to generate bogus software, the proposed system analyzes and obfuscates original source code to generate bogus programs that are compilable, realistic looking and dissimilar syntactically to the original program, through *static obfuscation*. Second, to detect or to track software exfiltration, the generated bogus programs include "beacons" that serve as a trap-based defense system. The proposed system designs various types of beacons to detect any possible unauthorized use of bogus software by providing various beacons for PDF, HTML as well as software itself.

Software decoys are composed of bogus programs that are designed to deceive adversaries. Bogus programs are synthesized by software that is automatically transformed from original source code, but designed to be dissimilar to the original. By using the bogus programs along with beacons, the proposed system aims to detect internal and external adversaries who exfiltrate proprietary (bogus) software. The experiments reported in this paper evaluate decoy properties of bogus software by using various metrics for similarity [17] and software complexity[7]. As a result, the bogus software has low similarity and analogous software complexity to the original software. As code obfuscation is performed several times, we can obtain bogus software that is completely different from the original software.

The proposed system makes several contributions over the typical advantages of decoy systems. First, we have developed a new software-based decoy system that integrates deception mechanisms

using software obfuscation and beaconing techniques. Second, we have designed fake (bogus) software by using code obfuscation techniques. Third, we have proposed a method to detect insiders based on bogus software instead of fake information, and track the use of the software decoy based on beacons. Lastly, we have implemented a software-decoy system and evaluated it through real open source projects that are popularly used in the real world.

The rest of the paper is organized as follows. Section 2 explains the desirable property of the software decoy system. Section 3 gives an overview of the proposed system and describes the detailed methods used to design software-based decoys. Section 4 describes the implementation of system prototypes and the results of that experiment. Lastly, section 5 reviews related work, and section 6 states the conclusion drawn from this paper.

2. SOFTWARE DECOY PROPERTIES

Software decoys should be designed carefully with the knowledge and capability of adversaries in mind. In addition, software decoys should have several properties for our goal, which are similar to document-based decoys [5]. Software decoys system should be inherently *enticing*. The decoy system must *detect* the exfiltration of bogus software that has been purposely planted in the system. To provide the means of detecting the exfiltration, the system we developed injects a *beacon* into the bogus software. In addition, the decoy system should be clearly *conspicuous* to adversaries. The bait, any bogus software, should be accessible and visible to adversaries and hence provided in a honeypot or a local system. Lastly, the decoy system has a large set of bogus programs from original projects that are different from every other one. That is, *variability* should provide a decoy system with a variety of attractive bogus programs. Along with these properties, software decoys have to satisfy additional core properties as follows.

- *Compilable* and *Executable*: The bogus programs should be compilable without any error. The programs should be also executable for a reasonable amount of time so that the decoy can detect the software exfiltration and identify bogus software. The program that is to be successfully compiled should be run to produce a part of functions of the original software. These two properties are essential requirements to make the bogus software believable.

- *Indistinguishable*: An adversary should not be able to recognize whether a bogus program has been transformed from particular source code or not. The adversary should have great difficulty in distinguishing bogus programs from a lot of other source code. In other words, we should produce an unbounded collection of distinct and variable bogus programs. This property is crucial so that adversaries cannot easily determine whether particular software is fake, nor that it is a derivative of open source software, non-proprietary project.

- *Believable*: The transformed program should logically look like a normal program. This property makes adversaries trust it as if the bogus software were true and real source code. While in the process of transforming an original seed program, we should try to maintain the original program structure and keep logical control flow so that the bogus software look likes real runnable source code.

We will show in Section 4 that additional bogus software properties can be validated through extensive experiments with real open source projects. Widely accepted software metrics [17, 7], such as similarity and software complexity, provide evidence of the practicality of our proposed bogus software generator.

3. SOFTWARE-BASED DECOY SYSTEM ARCHITECTURE

This section will provide an overview of the system architecture that we designed and implemented to create a software-based decoy system. The system depicted Figure 1 is given an original software project including several programs (in Java) as an input seed. It then produces a bogus project having a series of bogus programs. There are three requisite processes to create the software decoy: program analysis, code obfuscator, and program generator.

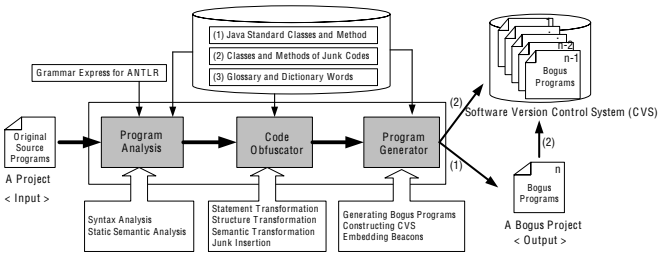


Figure 1: Software-based Decoy System Architecture

3.1 Analyzing Source Code

For any given input project seeding the synthesis of a software decoy, the proposed system first analyzes the syntax and the structure of each program in the project as in Figure 1. ANTLR (Another Tool for Language Recognition) [14] was used to extract information about syntax and the static semantics of each program. ANTLR is a parser generator with $LL(*)$ based on a context-free grammar argumented with syntactic and semantic predicates [15]. The current prototype targets Java-based projects, but the proposed system is easily extended because ANTLR provides a flexible and language-agnostic grammar development environment.

As for program analysis of a target project, we obtain information about what classes/variables/methods are defined, how they are related and where they are used. The extracted information includes (1) *package* declaration and *import* information, (2) *class* and *interface* names, (3) *member variables* names and types, (4) *member method* names, types, and parameter information, (5) mapping between *package* and *class/interface*, (6) mapping between *member variables* and *method*, (7) mapping between *class* and *interface*, and (8) the scope of *local variables*. The information is significant because when source code is transformed from one to another, other places matched with or related to the code must be consistently modified in order to make the overall project compilable and runnable.

Finally, as in Figure 1, through program analysis, we create two databases that are used to generate a bogus project. First, we analyze the syntax of Java standard APIs to generate a database of standard *class*es and *method*s. This database is important to obfuscate a target program carefully since the APIs should be mostly preserved during code transformation. Second, we extract sample *class*es and *method*s from Java sample source code collected from the Internet. This database is utilized to insert junk code in obfuscating target programs.

3.2 Obfuscating Source Code

After determining the syntax and the structure of target programs, the code obfuscator transforms original programs into bogus programs by making thorough changes in the form, syntax, or semantics of the original programs. This is called a *code transformation* as Definition 1. The proposed system modifies the semantics of the program slightly while the program is being continuously transformed. The proposed system has four different code transformation methods as follows. All the code transformation methods are closely related and the effects are interchangeably affected in programs since relevant statements should be changed together.

1. *Statement Transformation*: This transformation renames all the variables and methods for each statement in a program. Based on syntax information from program analysis, it alters the name for all *class*es, *method*s, and *variable*s in an original program. When changing all the names, the associated statements for variables, methods, and classes should be automatically renamed in all of the programs in a given project. The statement transformation replaces the original names for *class*es, *method*s and *variable*s with bogus ones. When changing the names of *class*es and *method*s, the bogus names are selected in the database of glossary and dictionary words, as in Figure 1, according to user-defined themes, such as shopping-related, health-related, financial-related software, etc. This is a basic code transformation before applying other code transformations.

2. *Structure Transformation*: A program is structured in different lines in order to be more readable, but it does not have strict and firm rules. The original structure of a program can be changed in various ways: (1) reordering primitives and methods, (2) breaking abstractions, (3) expression change, (4) control structure modification, and (5) changing data types.

   First, we can randomize the placement of as many modules within a program, methods within a module, and statements within a method as possible. Second, by reconstructing new packages and modules, it breaks the original abstraction of a program, which thwart adversaries from understanding the original target program. Third, the proposed system replaces operators, such as assignment, multiplication, and comparison, into different expressions. There are an arbitrary number of ways to turn a given arithmetic expression into a sequence of different elementary statements. For example, multiplication by a constant is often turned into a sequence of less obvious *add*s and *shift*s. Fourth, the control structures in a program can be used interchangeably to alter the structure of a program. The control structures include a conditional statement(e.g. if or else), a loop statement, (e.g. for, while), a selective statement(e.g. switch), and a jump statement(e.g. goto, continue, break). Lastly, data types in functions' parameters and variables are also changed if possible.

3. *Junk Code Insertion*: Bogus programs are diversified while generated in different ways by inserting any junk code as additional parts in a program. To insert junk code, there are several possible methods: (1) dead code insertion, (2) redundant statements, (3) method injection, and (4) code copy.

   First, the proposed system can add any number of blocks that can never be executed, such as *class*es, *method*s, etc. These are called *dead code*. Second, we place irrelevant or relevant statements for each line of a program. For instance, another variable or constant value can be declared and the variables are used any place in a program. Third, the proposed system clones bits and pieces of different *methods* in any given program, and the copied code looks different from the original one as a result of the code transformation, such as renaming, changing parameters in a method, etc. Lastly, from the database of *class*es and *method*s for junk code as in Figure 1, the proposed system selects one of them and reuses an arbitrarily chosen part of the code to generate bogus programs.

4. *Semantic Transformation*: The semantics of original programs can be also changed in different ways. First, the control flow of a program is naturally obfuscated while performing the proposed code transformations. Second, through *call* graph modification, the semantics of an original program can be changed. Specifically, the use of inserted methods and inserted code blocks first tweak an original call graph.

DEFINITION 1. *Let $T : \mathbb{P} \rightarrow \mathbb{P}'$ be transformation from program to program. T is code obfuscation, where $P_B = T(P_o)$ has a part of functions of P_O. T is a set of specific transformation elements, $t_1, t_2, \ldots t_n$. We enumerate several transformation techniques above. There are many other transformations possible, but what we have designed is sufficient for a proof of concept demonstration.*

The generated bogus program, P_B, should be different from the original source program, P_o to make it *indistinguishable* from the seed source program. The two programs can be evaluated according to two metrics: software *similarity* and *containment*. *Similarity* Δ is able to determine if two programs are very similar. Since the two programs, P_o and P_B, should be very dissimilar, the similarity should be less than a threshold λ as in Eq. (1).

$$\Delta(P_o, P_B) < \lambda \tag{1}$$

Containment Θ evaluates if one program is partially contained in another. Because the transformed bogus program P_B should have very small parts of code of the original source program P_o, the *containment* should be less than a threshold β as in Eq. (3.2).

$$\Theta(P_o, P_B) = \frac{\#\ of\ lines\ matched\ between\ P_B\ and\ P_O}{Total\ \#\ of\ lines\ in\ P_O} < \beta$$

The *Similarity* Δ of two programs is a number between 0 and 1, such that when the similarity is close to 1, it is likely that the two programs will be approximately the same. Similarly, the containment Θ of P_B in P_o is a number between 0 and 1 that, when close to 1, indicates that P_B is approximately contained within P_o. Section 4 shows that the generated bogus programs are completely different from the seed programs through these metrics.

As explained above, there are many different techniques for code transformation. The current system stops generating targeted bogus software when the similarity falls below a predefined threshold. While transforming, Java standard libraries, keywords and reserved words should be preserved.

3.3 Generating Bogus Programs for a Project

Based on the code transformation methods, the proposed system generates an arbitrary amount of decoy (bogus) software with any given input. The following outline below explains the method to generate a large number of different programs or diverse versions of similar programs. First, for any given input project, the proposed system generates different bogus software programs either from the original software or from the bogus software. Second, from the bogus software, the system produces a series of similar bogus programs so that software version control systems maintain a chain of history for the original project.

1. Generating different bogus software

   - From an original software:
     $P_O \overset{T_j}{\rightarrow} P^k_{B_n}$ (Note that $T_j = \{t_1, t_2, ..., t_n\}$, $i, j, k = \{1, ..., n\}$ and t_i is an specific transformation in T)
   - From previous bogus software:
     $P^k_{B_n} \overset{T_j}{\rightarrow} P^l_{B_n}$ (Note that $T_j = \{t_1, t_2, ..., t_n\}$ and $i, j, k, l = \{1, ..., n\}$ and t_i is an element in T)

2. Generating various versions from the bogus software for the CVS repository
   $CVS(P^m_{B_n}) \overset{t_n}{\rightarrow} CVS(P^m_{B_{n-1}}) \overset{t_{n-1}}{\rightarrow} \ldots CVS(P^m_{B_i}) \cdots \overset{t_1}{\rightarrow} CVS(P^m_{B_1})$
   (Note that t_i is an element in T, m=l or k, and $i, l, k = \{1, ..., n\}$)

Looking at the first step in more detail, the proposed system creates a variety of decoy software from original source code. Each resulting bogus software is different from every other one. In addition, the system uses previous bogus software to generate other new and different bogus software. The resulting bogus programs are dissimilar to each other depending on the number of iterations of code obfuscation (T). For any given input, the code transformation produces different kinds of decoy programs that are less than a predefined threshold of similarity and containment.

Second, a project is managed by software version control systems, such as subversion, git, etc., to keep updating new source code and tracking different software versions. To make decoy software realistic, the bogus software should be maintained to look like a real project by using one of the software version control systems. We generate a series of different versions from the first resulting bogus software under the CVS version control system. Specifically, the code transformation(T) has a set of different elements, $t_1, t_2, ..., t_n$. One element of the transformation method, t_i, is selected to generate slightly different versions of the bogus program every time.

3.4 Embedding Beacons

Each bogus program has a stealthy beacon that provides a signal indicating when and where the particular bogus program was used. The beacon plays a valuable role in identifying the exfiltration of software, distinguishing between bogus programs and original programs, by throwing an alert to a server for *detectability* of the decoy. The proposed system designs threes different types of beacons in a bogus project: PDF-based beacons, HTML-based beacons, and library-based beacons. The beacons report valuable information about an IP address, current date and time, a software version, etc.

First, typically, software provides several documents such as a README file for compiling and execution instructions, API descriptions and license information. For those documents, our proposed system adopts a technique to embed beacons into PDF, as proposed in [5]. In other words, we generate a PDF file by including any context in documents of a project and inject *Javascript* to send a signal to a server when the generated PDF file is opened.

Second, we utilize *javadoc* tool for generating API documentation in HTML format from Java source code. After creating the API documentation in HTML format, we embed *Javascript* with the generated HTML. When the documentation is opened, the embedded beacons make a signal to send the adversary information.

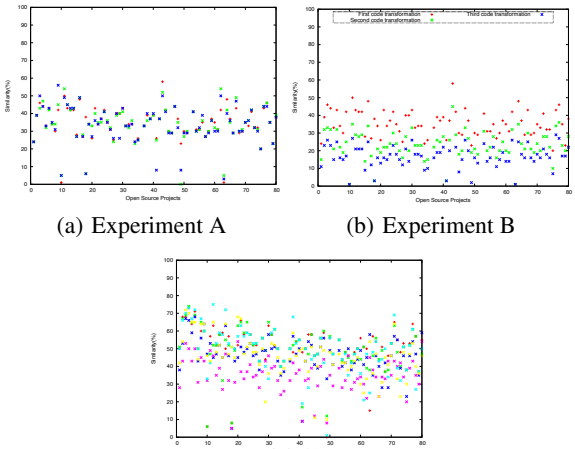
Lastly, for software embedded beacons, we can embed the code (libraries) that sends signals to a server upon program compilation or execution. The bogus program can be modified to use a library that must be downloaded in order to successfully compile it. Then, the request for the library on the server is a positive indication the bogus program is about to be compiled.

4. EVALUATION

We utilized Open Source Software (OSS) to evaluate the proposed system. As shown in Table 1, we evaluated 80 projects of the latest versions of OSS based on Java language. The OSS projects for this experiment were collected from the Apache Software Foundation and SourceForge [1]. Table 1 shows a summary of different file sizes for the collected OSS projects. Even though we collected Java-based OSS projects, a project can utilize different languages, such as XML, HTML, Python, and so on. For this experiment, we considered only Java-based source code files in each project.

File Size(F)	# of OSS Projects	Category	# of OSS Projects
F < 10MB	30	System/Build Tools	12
10MB ≤ F < 30MB	18	Financial/Shopping Programs	2
30MB ≤ F < 60MB	12	Health Programs	3
60MB ≤ F < 100MB	11	Content/Project Management Tools	17
100MB ≤ F < 500MB	7	Language Tools	6
500MB ≤ F < 1G	1	PDF Generating Program	5
1GB ≤ F	1	Web Applications	12
Total # of Projects	80	Miscellaneous	23

Table 1: The Collection of Open Source Software (OSS) Projects. A total of 80 OSS projects covering a variety of themes were studied.



(a) Experiment A (b) Experiment B

(c) Experiment C

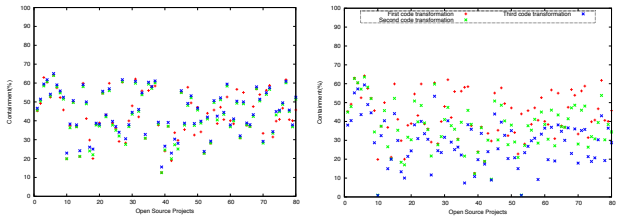
Figure 2: Similarity based on MOSS. (The x-axis is each of 80 open source projects.)

In the experiments, we evaluate various features of the proposed system as follows [2]

- Similarity and Containment: We evaluate the similarity and the containment between the bogus project and the targeted original project. We used well-known software plagiarism tools, MOSS [17]. This experiment shows that the generated bogus software is completely different from the original one.
- Software Complexity: Based on software metrics proposed in [7], we evaluate the software complexity of the resulting

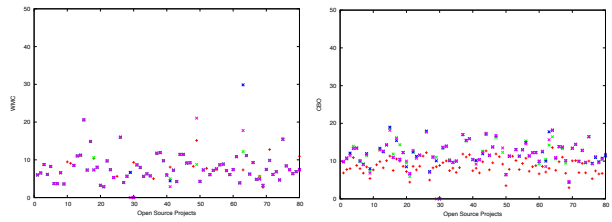
[1] http://www.apache.org/, http://sourceforge.net/

[2] For beacons, we tested all types of beacons with the collected OSS projects.

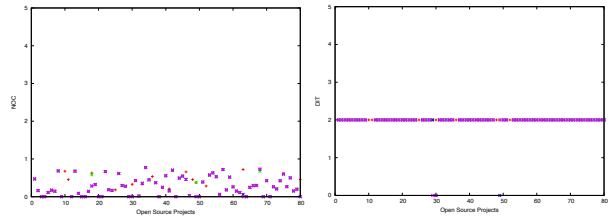


(a) Experiment A (b) Experiment B

Figure 3: Containment (The x-axis is each of 80 open source projects.)



(a) WMC(Number of methods defined in class) (b) CBO(Coupling between object classes)



(c) NOC(Number of immediate sub-classes of a class) (d) DIT(Maximum inheritance path from the class to the root class

Figure 4: Software Complexity for diverse bogus projects and an ordinal project (from Experiment A)

bogus projects. As original projects are transformed to create bogus programs, design weaknesses in the structure of the newly created bogus programs can make them vulnerable to detection by professional adversaries. Hence, we use software metrics [7] to evaluate the soundness of the design and structure of a bogus project so that the generated bogus software looks like normal source code. We utilize various metrics: Weighted Method per Class(WMC), Coupling Between Objects(CBO), Number of Children(NOC), and Depth of Inheritance Tree(DIT). These software metrics are widely accepted to evaluate software complexity.

We experimented with various combinations of bogus projects and original projects as follows (Note that T_j = $\{t_1, t_2, ..., t_n$ $\}$ and j, k, l = $\{1, ..., n\}$). These various experiments demonstrate that the proposed system creates a large number of dissimilar bogus projects for a given project. We executed code transformation for the experiments A and B.

1. **Experiment A:** With the same input, we produced *diverse* bogus projects. We transform an original project three times to generate different bogus projects. The input is always the same target original project. Thus, the output will be three different bogus projects from the same original project, $P^1_{B_n}$, $P^2_{B_n}$, $P^3_{B_n}$. (i.e. $P_O \overset{T_j}{\rightarrow} P^k_{B_n}$)

2. **Experiment B:** With different inputs, we generate different bogus projects. We transform an original project and its resulting bogus projects consecutively. In other words, the first input is an original project, the second input is the first resulting bogus project, the third input is the second resulting bogus project, and so on. The output is three different kinds of bogus projects from different inputs. We estimate the number of code transformation iterations to satisfy a desirable threshold. (i.e. $P_O \overset{T_j}{\rightarrow} P^k_{B_n} \overset{T_j}{\rightarrow} P^l_{B_n}$)

3. **Comparison C:** We evaluate the similarity among all the resulting bogus projects from Experiment A and Experiment B. (i.e. The comparison is a combination of $P^k_{B_n}$ and $P^l_{B_n}$)

Similarity: Figure 2 (a) shows that the proposed system drops from 45% to 60% of the similarities between bogus and original projects in the first iteration. This means that the proposed system dramatically changes given original software to bogus software. The bogus software seems totally different from the original software.

Figure 2 (b) shows the similarity after transforming one target project three consecutive times. The similarities between the first resulting bogus projects and the target original project are less than 50%. However, as we proceeded to perform code transformation several times, the similarities were decreased approximately 10% to 20% for each iteration. The similarity between the bogus project in the last iteration and the original project was in the range of 0%-30% for MOSS. This means that the proposed system can dramatically obfuscate original source code with only 2 or 3 iterations to obtain desirable similarity satisfying a predefined threshold.

Figure 2 (c) displays the diversity of bogus programs. Even though we used the same original project, the similarities among the resulting bogus projects are very different. In addition, through the course of several iterations, the similarities among the different bogus projects become low.

Based on our findings, we expect that the resulting bogus project will have low similarity so as to thwart adversaries and render them incapable of distinguishing bogus programs from real programs.

Containment: Figure 3 describes the containment between bogus projects and original projects. To measure it, we extracted the number of lines matched between two projects by using MOSS. We calculated the total number of lines in Java source code for each project. The results is very similar to the result of similarity, so we discuss these results in brief., Figure 3(a) showed that the first resulting bogus program had less than half of the original code. As in Figure 3(b), the containment between different bogus projects from one target project gradually decreases, according to the number of transformation iterations. The last bogus project from one target project includes a small portion of original source code. However, the containment is high between different versions of one bogus project for the version control system.

Software Complexity: Figure 4[3] shows the results concerning software complexity with respect to each of the four metrics: WMC, CBO, NOC, DIT. The experiment validates one of the system properties in Section 2 as being *believable*.

To achieve another property, that is, to be believable, the transformed bogus projects should have similar software complexity simulating real projects. Figures 4 shows that the complexity in the resulting bogus projects in our proposed system is in fact similar to the original projects. In the case of WMC, the bogus and original projects have small classes in the codes. The DIT is 2 in

[3]Due to the space limitation, we show only the result of Experiment A. However, the result of Experiment B is similar to Figure 4.

each case, since DIT should in general be less than 5. Since the CBO should generally be less than 14, the accumulated total for both projects is less than 20. Almost identical results are obtained in both projects when measuring NOC.

Therefore, even though the proposed system transforms source code, the resulting software complexity remains similar to the original project. We expect that adversaries can not help but interpret bogus projects as normal source code, failing to notice the forgeries despite their best efforts using these standard software measurement tools.

5. RELATED WORK

Decoy Technologies have been a critical defense method to secure our computing system. Cliff Stoll was the first to use decoy files and honeytokens to detect insider attacks exfiltrating information [18, 20]. Bell and Whaley proposed the structure of deception used to hide real information while exposing false information [4]. Bowen et al. designed an automated system for generating decoy documents [5, 6]. They also defined the desirable properties for decoy documents [5], and the proper deployment of decoy documents was suggested through various user studies [16]. These methods aimed to detect insider attacks based on fake information to confuse and deceive adversaries.

To create software decoys, we utilized various code obfuscation techniques that are used in order to protect software copyright. Fred Cohen [9] first proposed call obfuscation to generate syntactically different but semantically identical versions of the same program. There has been much work on program diversity through semantics-preserving transformations [3, 12, 11]. Such code obfuscation is also used in generating malicious programs, such as metamorphic or polymorphic codes to avoid virus scanners [8, 10]. In this paper, we utilized code obfuscation to automatically generate software decoys.

6. CONCLUSION

We proposed a software-based decoy system that is designed to identify software exfiltration by insiders through complete isolation of proprietary software from planted bogus software. The proposed system is a trap-based defensive system that is intended to deceive malicious adversaries, forcing them to expend considerable effort to differentiate real source code from bogus programs. To create software decoys, we utilized various code obfuscation techniques that are designed to protect a program from analysis and unwanted modification. Code obfuscation transforms a program either by inserting new code or modifying existing code. This makes the proposed system generate a large number of diverse believable bogus software programs from any given input program. To our knowledge, the proposed system is the first to study how to automatically generate large volumes of bait software, represented as ordinary normal source code project archives in a file system, that is believable and difficult it for an adversary to judge as fake.

7. REFERENCES

[1] http://sneakers.cs.columbia.edu:8080/fog/.

[2] DARPA-funded fake docs track unauthorized users. http://www.theverge.com/2011/11/4/2537647/darpa-fake-documents-security-wikileaks.

[3] B. Anckaert, M. Jakubowski, R. Venkatesan, and K. De Bosschere. Run-time randomization to mitigate tampering. In *Proceedings of the Security 2nd international conference on Advances in information and computer security*, pages 153–168, 2007.

[4] J. B. Bell and B. Whaley. *Cheating and deception*. Transaction Publishers, 1991.

[5] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *5th International ICST conference for Security and Privacy in Communication Networks (SecureComm)*, pages 51–70, 2009.

[6] B. M. Bowen, M. B. Salem, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Designing host and network sensors to mitigate the insider threat. *IEEE Security & Privacy*, 7(6):22–29, 2009.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transaction on Software Engingeering*, 20:476–493, June 1994.

[8] F. B. Cohen. Defense-in-depth against computer viruses. *Computers & Security*, 11(6):563–579, 1992.

[9] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[10] F. B. Cohen. *A Short Course on Computer Viruses*. John Wiley & Sons, 1994.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Aukland, 1997.

[12] K. Heffner and C. S. Collberg. The obfuscation executive. In *Proceedings of 7th International Information Security Conference(ISC)*, volume 3225 of *Lecture Notes in Computer Science*, pages 428–440, September 2004.

[13] M. A. McQueen and W. F. Boyer. Deception used for cyber defense of control systems. In *Proceedings of the 2nd conference on Human System Interactions*, HSI'09, 2009.

[14] T. Parr and K. Fisher. Ll(*): the foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation(PLDI '11)*, 2011.

[15] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to ll(k): pred-ll(k). In *International Conference on Compiler Constrution (CC)*, pages 263–277. Springer-Verlag, 1994.

[16] M. B. Salem and S. J. Stolfo. Decoy document deployment for effective masquerade attack detection. In *Proceedings of the Eighth Conference on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA)*, July 2011.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85, 2003.

[18] L. Spitzner. Honeytokens: The other honeypot. 2003.

[19] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. 5:26–40, 2006.

[20] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop 2004*, (June):116–122, 2004.