Enhancing Honeypot Stealthiness

by

Ng, Jun Ping

B. Computing (Computer Science)
National University of Singapore, 2005

SUBMITTED TO THE SMA OFFICE IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE
AT THE
SINGAPORE-MIT ALLIANCE

June 2006

Signature of author: _____

CS Programme
June 2006

Certified by: _____

Mr Chua Kuan Seah
Dissertation Supervisor

Certified by: _____

Assoc. Prof. Wong Weng Fai
Dissertation Supervisor

Accepted by: _____

Assoc. Prof. Ng Hwee Tou
Programme Co-Chair, CS Programme

Accepted by: _____

Prof. Tomas Lozano-Perez
Programme Co-Chair, CS Programme

Acknowledgements

I take this chance to thank Kuan Seah for giving me the opportunity to work on this interesting project, and for the trust and freedom vested in me throughout.

I also like to express my heartfelt appreciation to Dr. Joseph Lee for his patient guidance and input. Joseph seeded many of the ideas developed in this thesis.

Part of the implementation of the system developed in here is credited to Alvin Cai, who built the decision engine component of the Behavior Engine. Alvin also reviewed the draft of the grammar for BeeHive, and offered several useful suggestions to enhance it.

Contents

Enhancing Honeypot Stealthiness

by

Ng, Jun Ping

Submitted to the SMA office on 9 May 2006
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Abstract

Honeypots are useful tools that help administrators and researchers learn about network attacks and exploits, or even gather evidence against attackers for eventual legal prosecution. However because no real human user work on honeypots, there is a lack of physical and network activity on the honeypots. This is a potential give-away of the honeypots' identity to attackers. This research addresses this issue by proposing an autonomous system that can generate physical and network activity as part of a coherent simulation of human users working on the honeypots. Attackers relying on the activities performed on workstations to detect honeypots will be led to believe that they are intruding into a real production system.

Keywords:

Security, Honeypot, Compiler

Dissertation Supervisors:

1. Mr Chua Kuan Seah, Programme Manager, DSO National Laboratories
2. Assoc. Prof. Wong Weng Fai, Fellow, Singapore-MIT Alliance

“Suppose,”, he [Winnie the Pooh] said to Piglet, “you wanted to catch me, how would you do it?”

“Well,”, said Piglet, “I should do it like this : I should make a trap, and I should put a jar of honey in the trap, and you would smell it, and you would go in after it, and ... ”

A. A. Milne: Winnie the Pooh

Chapter 1

Introduction

1.1 Problem Statement

This project works on concealing a honeypot's true identity and purpose from an attacker.

The approach taken is to have the honeypot autonomously execute meaningful mouse movements and keystrokes. Through this behavior emulation on the honeypot, we want to convince an attacker observing the honeypot that a human user is working on the machine, and lead him to believe that this is a real production system.

1.2 Motivation

Honeypots are used to silently observe and record an attacker's actions upon penetrating the system. This information is useful to help us learn more about the tools and techniques the attacker employ, and even form the basis of evidence for future legal prosecution.

To carry out their function, honeypots need to hide their true identity and purpose from the attacker. Once the attacker realises that he has broken into a honeypot, it is almost certain that he will leave the system. We thus want to have the honeypot appear as genuine a production system as possible to the attacker so

as to keep his attention.

Current honeypots mainly comprise unused systems loaded with system software and applications. Well-executed honeypots are usually isolated from the real production network for security reasons. To an attacker breaking into such systems, the lack of network traffic within the honeynet as well as the lack of physical activity on the honeypot itself is a dead give-away.

1.3 Aims

This project thus seeks to incorporate an autonomous behavior engine in honeypots which will generate meaningful physical and network activity, to help conceal the honeypots' true mission from attackers.

The objectives for the project include :

- Developing a behavior engine that will select meaningful tasks to perform
- Designing a language to allow administrators to customise the generated behavior of said behavior engine
- Automating the translation of the new language into modules within the behavior engine

1.4 Organization

Following this chapter, Chapter 2 explains the concept of honeypots and touches on some common implementations of honeypots. Then in Chapter 3, we will go through the details of the various components of the behavior emulating platform developed in this thesis. Chapter 4 evaluates the effectiveness of our platform in reference to the project objectives, and Chapter 5 rounds up with a look at possible extensions to this piece of work.

Chapter 2

A Honeypots Primer

In this chapter, we will go through some background information relating to domain of this project — honeypots. We will explain what honeypots are about and take a quick look at some common implementations. Further, we will also discuss some techniques that attackers can employ to detect honeypots.

2.1 Honeypots

There has been lively discussions among professionals on the proper definitions for a honeypot. The functional definition presented by L. Spitzner [13] is general, but effectively summarizes the key purpose of honeypots.

> A honeypot is a resource whose value is in being attacked or compromised. This means, that a honeypot is expected to get probed, attacked and potentially exploited. Honeypots do not fix anything. They provide us with additional, valuable information.

What then exactly are honeypots? Honeypots are really a flexible tool which takes on various forms. For example, a honeypot can be just simply be a workstation running software emulating network services like a **FTP** server. Incoming connections to the network services are logged and studied. At the other end of the complexity spectrum, a honeypot can also be a complete, proper workstation with

a real operating system hosting real network services.

To facilitate discussion, researchers have classed the first type of honeypots as low-interaction honeypots, and the latter as high-interaction honeypots. Low-interaction honeypots by nature offer attackers limited options and flexibility. They are easy to manage and set up, but are also dis-interesting to the seasoned attacker. A few common implementations for low-interaction honeypots include *honeyd* (see next section) and *BackOfficer Friendly*.

High-interaction honeypots on the other hand offer real systems for attackers to interact with. This allow us to monitor and learn about the full extent of attackers' behavior. However, high-interaction honeypots are double-edged swords as attackers may use them to exploit real production systems. These honeypots thus require more effort to deploy and maintain. Typical implementations include running *Linux* in user mode, or loading virtual machines like *VMWare*. These implementations are preferred because they offer the honeypot maintainer more control than possible with the original operating system installation. To mitigate the potential risks of these high-interaction honeypots, it is also common to sandbox these environments with mechanisms like *jail* which impose constraints on the hosted virtual environment.

The underlying theme behind all honeypots however is to attract attacks and maintain extensive logs during the intrusion for later study and analysis. From the logs we can learn about the techniques and tools used by the attackers, and their purpose and goal. *Sebek*[10] by the Honeynet Project[3] is one example of a data capture tool specially built to capture such intrusion activities.

But while we will like the honeypot to be probed, intruded, and hacked into, attackers want to avoid honeypots for various reasons. Hacking into a system that holds no real value, and having every of their actions and maneuvers logged and recorded makes for a lot of unfruitful work. The tools and exploits used by the attackers also lose their value once they become public knowledge, so it makes sense not to have them tracked.

The goal of the honeypot being to observe and track the attacker, it is thus of paramount importance that the honeypot's identity remain hidden from the

attacker for it to be successful.

2.2 Common Implementations

Let us now take a look at some common implementations for honeypots to get a better idea of what they are. Of the numerous ways available to set up a honeypot, we will examine *honeyd*, an example of a low-interaction honeypot, and then *VMWare*, a virtual machine software that emulates x86 hardware.

2.2.1 Low Interaction Honeypot — honeyd

An example of a low-interaction honeypot is the open source *honeyd*[2]. honeyd runs as a daemon on a host machine, and emulates network services such as a **FTP** or **SMTP** server. Honeyd can also create virtual hosts on the network. This means that honeyd will accept incoming network connections directed at IP addresses that do not exist in its same subnet. Users can have honeyd emulate various network services on the virtual hosts. Attacker interaction with these emulated services are logged for later study.

The support for virtual hosts is useful because in a properly configured production network, no traffic is expected to be sent to the virtual IP addresses. Any connections made to the virtual IP addresses thus indicate with high probability an intrusion. This gives us a very low false positive rate.

Though there is not security by obscurity[11], the popularity and open-source nature of honeyd had prompted many studies into it. From these studies, attackers have been able to fingerprint particular honeyd implementations for example, making it possible to detect systems running as honeypots. Nontheless honeyd is currently well-maintained and regularly updated, thus keeping its edge in evading detection.

2.2.2 High Interaction Honeypot — VMWare

VMWare is a software emulator for x86 hardware. Thus with VMWare we can install guest operating systems on top of an existing one (termed the host system).

This makes it very useful for creating honeypots of various operating systems as VMWare is able to run the different versions of Windows, Linux and the BSD family of Unix.

Attackers intruding into such a system have a complete operating system waiting for him, but yet which is sandboxed and segregated from the actual machine. This gives operators more control over security. Since a complete operating system is emulated, operators also get a flexible honeypot implementation as a full set of software and tools on the original operating systems can be used.

There is usually a need for such instances of high interaction honeypots to install additional checks and constraints so that the payload of any intrusion is limited to the honeypot or honeynet itself. Effort needs to be undertaken for example to restrict outgoing network traffic from the honeynet to prevent it from being abused as a launchpad for further attacks into other systems.

2.3 Detecting Honeypots

Looking at the descriptions of the honeypots so far, it may seem that from the point of view of the attacker, it will be hard to distinguish between a real production system and a honeypot. With a VMWare honeypot for example, the attacker gets the full operating system to intrude into, and all the tools and exploits the attacker may use will work properly.

However, this is not the case. The emerging prominence of honeypots has led to a closer scrutiny of the the various implementations, and attackers have found ways to help them decide if they have broken into a honeypot. For example, the same problem that plague software development all these years surfaced in honeyd too. Buggy implementation in honeyd had it reassemble invalid TCP SYN packet fragments which contain different protocol numbers[9]. On re-assembling the message, honeyd then erroneously replies to it with a SYN/ACK message. This bug is reasonably unique to honeyd, and allows attackers to fingerprint honeyd IP addresses in a target network.

Holz, T and Raynal, F in [1] also discussed techniques attackers can employ

to uncover high-interaction honeypots such as those deploying VMWare. This can be done by looking at the media access control (MAC) addresses of the machines' network interfaces. The MAC addresses of machines running VMWare fall into three specific ranges assigned to VMWare Inc. by IEEE[4].

With increasing end-user bandwidth capacity in recent years, it is also becoming feasible for attackers to establish covert channels to tunnel keystrokes and screenshots of the compromised systems back to their own machines. These can be used to observe how a workstation is being used. A lack of meaningful intelligent actions on machines also hint to the attackers that they have intruded into a honeypot.

2.4 Rat Race

So much like the many police-and-thief stories that are so prevalent, researchers and attackers are constantly pitting their wits against one another. On the one hand, honeypot technology is being improved to help honeypots stay covert and apt at their work. On the other, attackers are constantly updating their knowledge and developing new techniques and exploits to stay one up.

Chapter 3

Solution

The study into honeypots revolves primarily around improving data capture capabilities and detection evasion. Work targeted at the latter span a wide area, including :

1. improving on emulated network services so that the interaction with the attackers is made more realistic
2. improving on the delivery platform to make the guest honeypot systems less visible
3. balancing between the contradictory needs to sandbox the honeynets and to present a realistic network to the attackers

We have explained previously that one of the advantages of a honeypot is that it gives a very low false positive rate. We will not usually expect network traffic on the honeypot, and thus any signs of such traffic is an indicator of possible intrusion.

This very advantage however can be turned into its disadvantage. An attacker can elect to observe the activity that goes on in the honeypot and the corresponding honeynet. Since it is not likely for a real production system to be left unused over a period of time, this gives away the honeypots' identity away.

It seems that up to now, little attention has been paid to this problem. The lack of physical and network activity on honeypots is a big loophole which needs

to be plugged. To address this problem, one complete solution can involve several modules working in tandem. An instance of such a solution (summarized in Figure 3.1) may include :

1. A study into behavioral patterns and habits of real human users whose actions we want the solution to model and simulate.
2. Translating results of the study into some form of input script
3. An engine that takes an input script and decides the task to simulate at a point in time
4. An actuator that performs the actual keyboard events and mouse clicks according to the direction of the engine

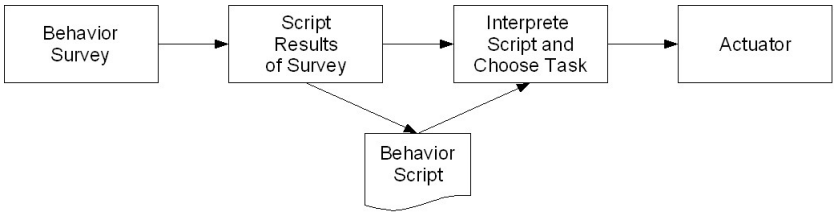

Figure 3.1: Various modules in a complete solution.

The engine and actuator will be loaded into a honeypot and they will enact the tasks that were captured during the behavioral pattern study. These translate into physical keystrokes and mouse movements and clicks on the honeypots, and these acts can in turn generate network activity (such as through surfing the web or sending emails for example). Attackers intruding into the systems see the generated activity and are assured that they are not on a honeypot.

3.1 Scope

Focusing on a part of the solution described above, the contribution of this thesis is in designing a language for coding the behavioral script and developing the interpretation engine.

3.2 Challenges

3.2.1 Scripting Language

To succeed in convincing the attackers, the *script* used to dictate the tasks to perform cannot be static. Static scripts will replay the same tasks after a while and attackers observing the honeypot for some extended period of time will be able to spot this peculiarity.

On the other hand, the script cannot be purely random and dynamic. The attacker may have some notion of the use of workstation he has intruded into, and the generated activity should match this. For example, an attacker intruding into what he perceived as a network in an administrative office setting would expect to see activity and traffic corresponding to such a scenario. The usage patterns of a secretary's workstation would not be the same as that of an engineer.

3.2.2 Engine

As the engine will be deployed on the honeypot, deployment and performance are the two main challenges it will face. Deployment should be easy and quick, particularly the interface with the scripting language and the actuator. Performance wise, the engine should be as invisible as possible to avoid detection.

There is an added technicality of concealing the engine process from an attacker, but this is outside of the scope of this thesis.

3.3 Goals and Objectives

Bearing in mind the challenges laid out previously, the scripting language should cater for dynamism and variation. It should also be flexible enough to accommodate most if not all desired behaviors to be enacted.

The engine will be required to support the constructs specified in the script. It should also have a small memory footprint as well as impose only a light processor load because it will run as a clandestine process in the honeypot.

The integration of the script and the engine should also require only minimal operator intervention to ease the deployment of the engine into honeypots.

3.4 Overview

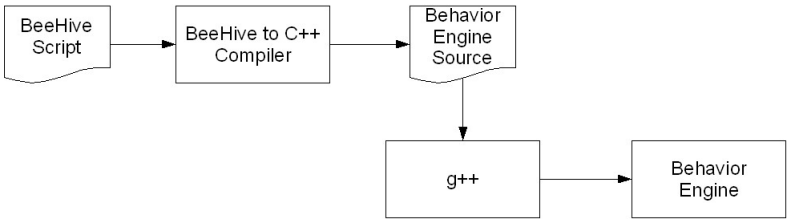

Figure 3.2: Overview of thesis system.

Figure 3.2 shows an overview of the system developed. **BeeHive** is the scripting language that is designed. A compiler translates a BeeHive script into the C++ source code of the **Behavior Engine**. After compilation with a C++ compiler like **g++**, we get the Behavior Engine.

Though the Behavior Engine is a product of the BeeHive compiler, it is in itself a full system that was separately engineered before integrating into the compiler.

In the ensuing sections, we will take a closer look at each of the key parts of the system, namely BeeHive, the Behavior Engine, and the BeeHive compiler.

3.5 BeeHive

As stated previously, two important considerations when designing BeeHive are dynamism and flexibility.

Dynamism means that there must be some way to specify task variation within the script, and not end up doing the same thing over and over again, while flexibility gives us allowance to specify any behavior we may need to.

The syntax for BeeHive is also engineered to be more English-like to cater for a less technically inclined audience.

The ensuing sections will highlight the key features of the language. As a reference, the complete BNF for the BeeHive grammar can be found in Appendix A.

A user guide has also been produced for BeeHive[6].

3.5.1 Event and Time Driven

To achieve dynamism and variation in BeeHive, a primarily event-driven approach is used. Referring to lines 2 to 15 in Listing 3.1, we can specify the actions to carry out when a new email arrives. Lines 18 to 25 also show that we can tag along actions to other actions.

This approach greatly facilitates interaction between individual honeypots as we will not have to specify the exact time we want to send an email reply for example, as would be the case for a time-driven approach.

Listing 3.1: Snippet of BeeHive script.

```
1
2  // Event-driven -> Responding to the new email event
3  CREATE EVENT new_email {
4    DO {
5      SEQUENCE {
6        WITH 80% CHANCE outlook_read_new_email()
7
8        // Nested specifiers
9        WITH 20% CHANCE CHOOSE 1 OF {
10         WITH 80% CHANCE AND 50% INTERRUPTABILITY
               outlook_reply_email()
11         WITH 20% CHANCE AND 0% INTERRUPTABILITY
               outlook_delete_email()
12       }
13     }
14   }
15 }
16
17 // Event-driven -> Tagging on actions to other actions
18 AT ACTION edit_word_document {
19   DO {
20     PARALLEL {
21       WITH 80% INTERRUPTABILITY surf_the_web()
22       WITH 80% INTERRUPTABILITY web_search()
23     }
24   }
25 }
26
27 // Time-driven -> Specifying important time markers
28 AT TIME morningBreak {
29   DO {
30     SEQUENCE {
```

```
31          WITH 100% CHANCE INTERRUPT
32        }
33    }
34  }
```

The benefits of an event-driven approach notwithstanding, time-driven actions are also supported to increase the flexibility of BeeHive. This is shown in lines 27 to 34 for example, which says that we will interrupt the actuator at the *morningBreak* time mark.

The resulting script is thus a "guideline" on what the Behavior Engine can choose to do as much as it is a "script" that specifies plainly what must be done.

3.5.2 Random Variations

There is extensive use of probabilistic statements within BeeHive to encourage variation in the behavior the Behavior Engine exhibits. For example, the *morningBreak* time mark is declared as shown in Listing 3.2. It is declared as a value sample from a normal distribution with mean at 1000h, and a variance of 15 minutes. All other declared values within BeeHive (such as the duration of each action) can be specified similarly. This provision ensures that the actions performed on two different days are never the same.

Listing 3.2: Varying declared values.

```
1  SET TIME morningBreak TO BE _NORMAL_DISTRIBUTION(1000,15)
```

3.5.3 Nesting Specifiers for Flexibility

To group a set of actions together, there are 4 action specifiers in BeeHive. These include the **SEQUENCE**, **PARALLEL**, **CHOOSE** and **REPEAT** specifiers.

In a nutshell, sequence operations execute serially, one after another. Parallel operations behave much like context switching on a modern operating system, with tasks being carried out concurrently. Choose operations are essentially parallel

operations, except that only a subset of the operations are performed, while repeat operations involving executing action in sequence repeatedly.

The nesting of these specifiers allows a wide range of desired operations to be specified and greatly enhanced the flexibility of BeeHive. An example scenario is demonstrated in the event response in Listing 3.1. The nested specifiers allow us to specify that we want to either reply to an email, or delete it after reading it.

3.6 Behavior Engine

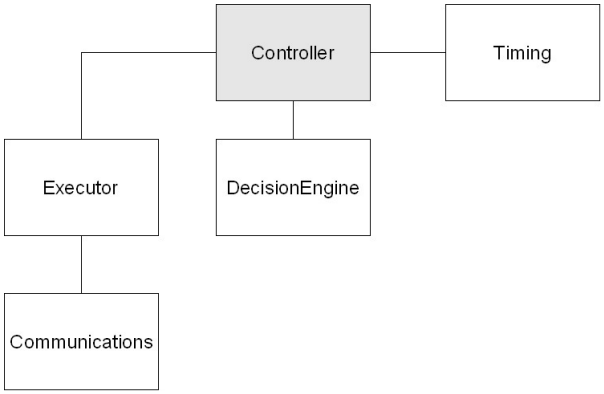

Figure 3.3: Design of Behavior Engine.

The key functionality of the Behavior Engine (BehEng) is to decide what activity to perform at each point in time. The decision will be based on the input BeeHive script to the system.

Figure 3.3 shows the different components that make up the BehEng. The *controller* is the entry point into the engine, and it is responsible for co-ordinating between the various modules. The *timing* module is responsible for system time updates and other time related functionality. Interacting with the controller, the *decision engine* is responsible for deciding and choosing the next tasks the Engine should perform. The controller will send tasks to be executed to the *executor*. This module takes care of the context switching between multiple tasks. It interfaces with the external actuator module through the *communications* class. The actual

communication with the actuator takes place over Windows sockets.

To better illustrate the control flow of the Behavior Engine, Figure 3.4 show the different threads active within an instance of the Engine, as well as the control flow within each thread.

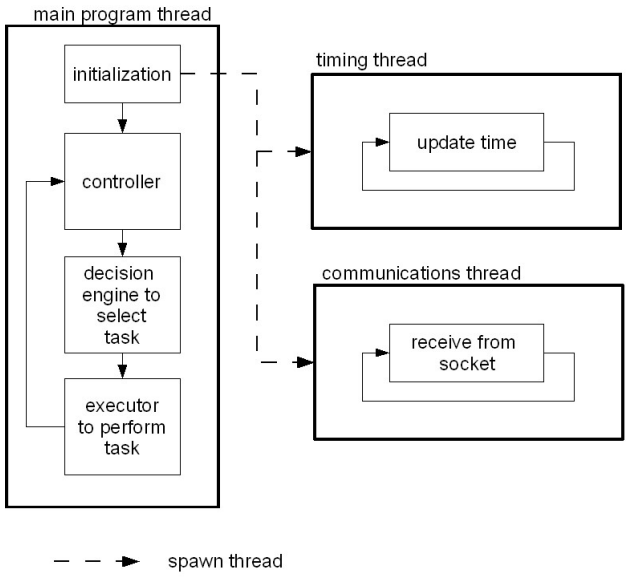

Figure 3.4: Threads and control flow.

The general framework of the system consists of these 3 threads — the main program thread, the timing thread, and the communications thread.

Within the main program thread, the controller calls on the decision engine to choose a task to execute. The decision engine is implemented as a priority queue, sorting on the urgency and priority of the eligible tasks. The selected task is then sent to the executor to be executed. More details on how the executor functions is given in the ensuing sub-section. After the task has been executed, the whole process is repeated.

The timing thread updates the internal time variable with the current system time. This update is performed once every minute, with the timing thread idle-waiting in between the updates.

The communications thread blocks on the receive call to the windows socket used to interface with the external Motor module. Each time new data arrives, this thread services the data, performs the necessary actions, and go back to block on the receive call.

3.6.1 Executor

As explained previously, tasks selected by the decision engine are sent to the executor for execution. The executor module is important because it belies the support for nesting of action specifiers, and concurrent execution of several tasks at the same time. A multi-threaded mechanism is employed to allow us to support these two important features.

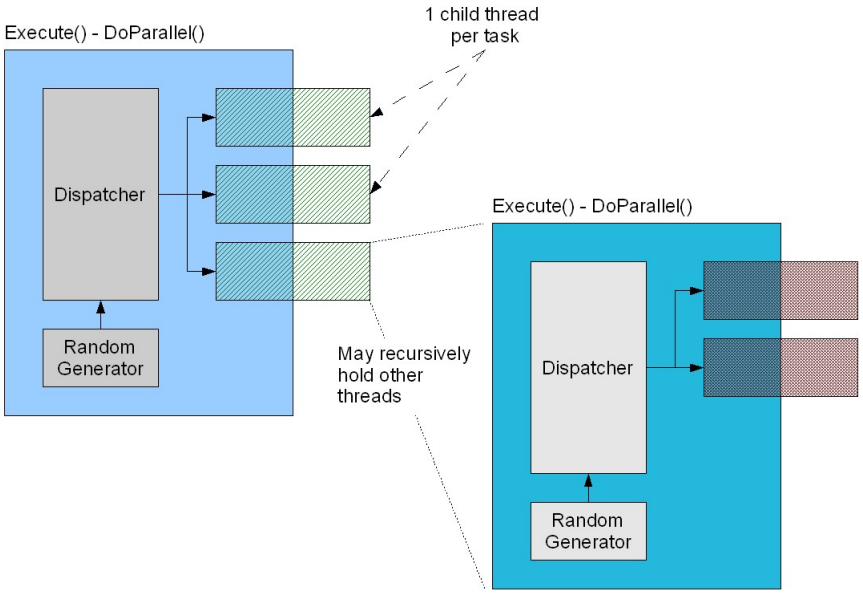

Figure 3.5: Multi-threaded execution mechanism.

Figure 3.5 shows how the execution mechanism works. When presented with a set of concurrent tasks, a dispatcher thread spawns several child threads to process each task. This dispatcher thread handles the pre-emption and context switching between the child threads. With this mechanism, we are able to simulate performing several tasks concurrently, as a real user would.

Because a different handler routine is used to process each different specifier, nesting of the action specifiers is achieved by having the child threads run the correct routines. In the figure, we see an example of another parallel dispatcher running within a child thread.

3.7 Compiler

The linking up of the BehEng and the input BeeHive script is performed by a compiler. The BeeHive compiler takes in a BeeHive source file and translates it into the C++ source code of the BehEng.

Large parts of the BehEng will be independent of the input BeeHive. The compiler's role is then to patch up the rest of the engine with the components dependent on BeeHive. This will then result in a version of the Behavior Engine "customised" to a particular BeeHive script.

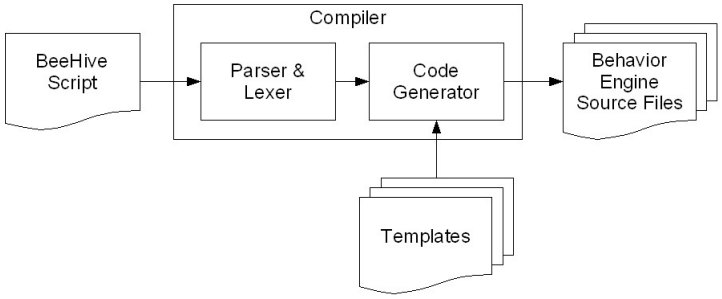

Figure 3.6: Overview of BeeHive compiler.

Figure 3.6 summarizes the compilation process. The compiler comprises of a parser/lexer, and a code generator. A semantic checker would have be an useful addition to the compiler, but is not included in the implementation of this prototype system.

To help integrate the BehEng into the compiler, a special set of template files are used. The code generator takes in a set of these template files and uses them to generate the BehEng source files. For the BehEng files that are independent of BeeHive input, the code generator simply replicate these files in its output. Template files which rely on BeeHive input carry special tokens which are picked

out by the code generator and replaced with the relevant code using information from the parse structure built by the parser/lexer.

For example, Listing 3.3 shows an excerpt of one of the template files.

Listing 3.3: Excerpt from a template file.

```
1 ...
2     static void addTimeStatements() {
3       %%time-statements%%
4     }
5 ...
```

Line 3 shows a special token `%%time-statements%%`. It will be expanded into what is shown in Listing 3.4. The resulting code is based on information extracted from the parse structure of the BeeHive input.

Listing 3.4: Replacing a special token from a template file.

```
1  ...
2      static void addTimeStatements() {
3
4        DataStructures::timeMarksTable.push_back(
5          new Time(string("startWork"),
6          ExternalFunctions::_Transform_To_Valid_Time(
7              ExternalFunctions::_NORMAL_DISTRIBUTION(830,20))
8                                  ));
9
10       ...
11     }
12 ...
```

Through the use of such a facility, the coupling between the code generator and BehEng is reduced. We can thus make small changes to generated BehEng files easily, without having to hack into the code generator.

3.8 Putting It Together

The three components explained above link up together to serve as a framework that allows us to give a "personality" to each honeypot. With this framework, we

can specify a desired behavioral trend for each honeypot, and have them "behave" according to these specifications.

With an appropriate specification, the honeypots can be expected to perform physical activity and generate network activity which is relatively consistent to what would have been if it had been a real production system. This gives us the capability to spoof attackers who may be observing the honeypot into believing that they are breaking into a real system.

In the bigger scheme of things, this framework can be part of a honeypot implementation alongside with traditional data capture tools like *Sebek* and intrusion detection utilities like *Snort*[12].

Chapter 4

Evaluation

The developed language and engine being part of a larger system, it is not possible to conduct a proper field test to evaluate how effectively our work contributes to the bigger objective of spoofing attackers about the identity of the honeypots. Hence for our context, the BeeHive language and Behavior Engine will be evaluated instead with the design goal and objectives as a yardstick.

4.1 BeeHive

Recall that BeeHive needs to accommodate dynamism and flexibility.

4.1.1 Dynamism

Dynamism is achieved through a combination of constructs that support randomized behavior. These include specifying values as samples from probabilistic distributions, and the use of probabilities in the various action specifiers. The former can be used to vary time marks as well as action durations and the latter can be used to inject variety in the actions to be carried out in response to events or time.

4.1.2 Flexibility

The expressiveness of BeeHive underscores its flexibility. Through support for nesting action specifiers, any combinations of actions can be obtained. Rather than a formal proof, we substantiate our claim with this intuition drawn from propositional calculus.

In propositional calculus, all propositional statements can be expressed in conjunctive normal form (CNF). In CNF, only the $\neg$, $\vee$ and $\wedge$ operators are used. Thus we can see that the $\neg$, $\vee$ and $\wedge$ operators are sufficient to express all possible propositional statements.

Consider a finite series of actions

$$Action_1, Action_2, ..., Action_n$$

We can view each $Action_i$ as a propositional variable as in predicate calculus. Then we can express a desired combination of actions to be performed as a propositional statement in conjunctive normal form (CNF). For example, we say

$$Action_1 \wedge (Action_2 \vee Action_3)...$$

to mean that we will do $Action_1$ and either of $Action_2$ or $Action_3$. The set of propositional statements obtained in this way will not make use of the $\neg$ operator.

To show that the action specifiers in BeeHive are sufficient to allow us to express any combination of actions, we are left with mapping the specifiers to the $\vee$ and $\wedge$ operators.

Depending on whether the order of executing the actions in such a conjunct is important, we can map either the **PARALLEL** or **SEQUENCE** specifiers to the $\wedge$ operator. This is because by definition, the $\wedge$ operator specifies that each conjunct will be executed. This is the same semantics for both the specifiers.

The **CHOOSE** specifier can be mapped to the $\vee$ operator. Any of the terms in a disjunct can be selected, or none at all, and this is similar semantics to the **CHOOSE** specifier.

We are then drawn to conclude that with the **PARALLEL**, **SEQUENCE** and **CHOOSE** specifiers, and by nesting them, we can sufficiently express any finite

series of actions in CNF. Support for an additional **REPEAT** specifier extends this claim to an infinite series of actions.

4.2 Behavior Engine

The technical objectives of the Behavior Engine are to keep its memory footprint small, and keep CPU requirements to a minimum.

For this, the implementation makes extensive use of native Windows thread synchronisation facilities like event signaling and idle waiting to cut down on redundant computation or active waits. Efficient data structures like hashtables are also used within the Behavior Engine to help achieve efficiency.

The compiled version of the resulting Behavior Engine is about 180Kb in size. The average load the engine imposes on the processor approaches 0%. This is obtained by averaging over samples taken at intervals of 1 min over a day.

From these we see that the engine manages to meet the targets of a small memory footprint and negligible processor load.

4.3 Sample Output

To give a better idea of how specifications in BeeHive translate to actions, Listing 4.1 and Listing 4.2 shows a snippet of a BeeHive script, and some sample output obtained by running the Behavior Engine for this script respectively.

The scenario specified in the script is that for the first half of a work day before lunch time. The user in this case is working on a word document, and surfs the web periodically for information, while keeping up to date with his web email account. During the morning break, no work is expected to be carried out.

Listing 4.1: Snippet of BeeHive script.

```
1 AT TIME startWork {
2   DO WITH urgency 10 AND priority 10 {
3     SEQUENCE {
4       WITH 100% CHANCE RESUME
5     }
```

```
6     }
7   }
8   AFTER TIME startWork {
9     ENSURE {
10      timeNow() < morningBreakStart
11    }
12    DO {
13      PARALLEL {
14        WITH 90% INTERRUPTABILITY edit_word_document()
15        WITH 30% INTERRUPTABILITY view_web_mail()
16        WITH 30% INTERRUPTABILITY web_search()
17      }
18    }
19  }
20  AFTER TIME startWork {
21    ENSURE {
22      timeNow() > morningBreakEnd
23    }
24    DO {
25      PARALLEL {
26        WITH 90% INTERRUPTABILITY edit_word_document()
27        WITH 70% INTERRUPTABILITY view_acrobat_document()
28        WITH 50% INTERRUPTABILITY web_search()
29      }
30    }
31  }
32  AT TIME morningBreakStart {
33    DO WITH urgency 10 AND priority 10 {
34      SEQUENCE {
35        WITH 100% CHANCE INTERRUPT
36      }
37    }
38  }
39  AT TIME morningBreakEnd {
40    DO WITH urgency 10 AND priority 10 {
41      SEQUENCE {
42        WITH 100% CHANCE RESUME
43      }
44    }
45  }
46  AT TIME lunchTimeStart {
47    DO WITH urgency 10 AND priority 10 {
48      SEQUENCE {
49        WITH 100% CHANCE INTERRUPT
50      }
51    }
52  }
```

Listing 4.2: Sample output from Behavior Engine.

```
1  At time 800 : system_resume
2  At time 800 : system_nothing
3  At time 801 : word_edit
4  At time 834 : web_mail
5  At time 835 : web_search
6  At time 843 : word_edit
7  At time 858 : web_mail
8  At time 905 : web_search
9  At time 912 : system_nothing
10 At time 912 : web_mail
11 At time 917 : web_search
12 At time 945 : word_edit
13 At time 1013 : system_nothing
14 At time 1013 : web_search
15 At time 1019 : system_interrupt      // Morning break starts
16 At time 1027 : system_resume         // Morning break end
17 At time 1027 : word_edit
18 At time 1059 : web_mail
19 At time 1100 : word_edit
20 At time 1101 : web_mail
21 At time 1105 : web_search
22 At time 1107 : system_nothing
23 At time 1110 : system_interrupt      // Lunch time
```

The sample output shows the inter-mixing of the word document editing (*word_edit*) action, web surfing (*web_search*) action, and the checking of a web email account (*web_mail*) action.

Before and after the morning break starts, special interrupt (*system_interrupt*) and resume (*system_resume*) actions ensure that the actuator will not perform any tasks within this period.

4.4 Limitations

We have seen that the framework developed in this thesis provides an efficient platform with which honeypots can exhibit desired behaviors. It is however not the only determining factor of success for the proposed behavior emulating system to help honeypots evade detection. For completeness, let us touch briefly on two other key factors. These include the modeling of user behavioral trends and habits, as well as the content generation capabilities of the actuator.

As stated previously, attacker may have a notion of the systems they are break-

ing into, and they will expect the compromised systems to exhibit physical and network activity consistent with this notion. If a honeypot is play acting as a secretary's workstation for example, it is not likely for it to surf to technological and scientific websites on a regular basis. Thus a study into user behaviors and habits will be most useful when preparing a BeeHive script for the Behavior Engine. This study can capture information such as the amortized average duration for each action, as well as important time marks such as lunch breaks and knock-off timings.

More importantly however, the content generated in composed emails and documents by the actuator will be subject to scrutiny by the attackers. The syntax and semantics of the content must thus not be illogical and irrelevant. The state-of-the-art in computer generated content is still lacking in this aspect, though this can be mitigated somewhat with clever use of pre-generated content within a database.

Advancements in these two factors will help maximise the potential of the framework we have developed.

Chapter 5

Conclusion

5.1 Further Work

Extending from this piece of work, an important enhancement to BeeHive is support for building "profiles" out of one BeeHive file. Generally, we can identify differernt groups of computer users we want to model and simulate, such as a group of engineers, secretaries, or managers. Within such a group of users, behavioral trends are highly similar. With such a co-relation between behavior for individuals within the group, it is uneconomical to have a BeeHive script for each identified user within the group.

We can instead build individual profiles from a BeeHive script. So the BeeHive script will code the general habits common to the group of users, while the individual profiles will incorporate individual preferences into such a BeeHive script.

As a concrete example, the BeeHive script for engineers would possibly involve actions relating to web searches and viewing of Word and Acrobat documents. However different engineers have different preferences for search engines. Instead of having a different BeeHive script for each engineer, it is better design to code a BeeHive profile script that corresponds to the activities of engineers, and then build from it several more specific BeeHive scripts incorporating the preferences of different individuals. Automating this process with a well-designed graphical user

interface will enable an operator to set up different behavior engines for various honeypots within a honeynet very quickly and easily.

Figure 5.1 summarizes the profiling process. No major grammar changes need to be introduced. Special tokens can be used within existing BeeHive scripts to represent variables for individual preferences. The profile customiser then acts as a pre-processor of sort to pick out these tokens and use information of individual preferences to output corresponding BeeHive scripts.

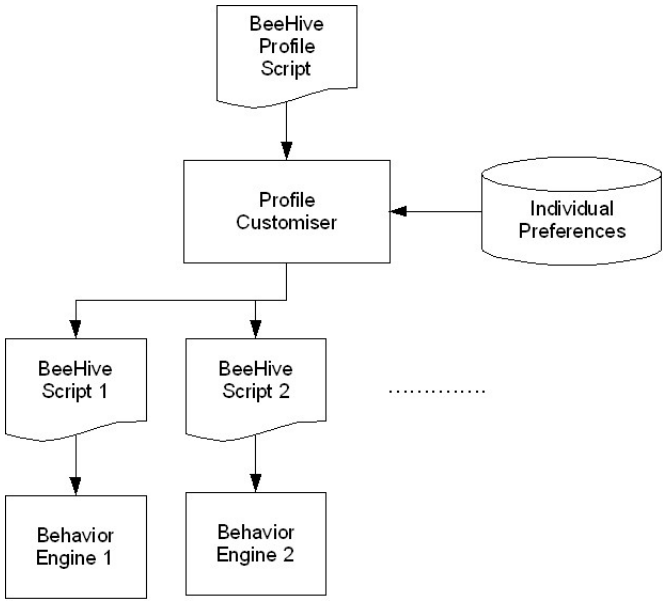

Figure 5.1: Overview of the profiling process.

5.2 Additional Documentation

There are three sets of documentation, one created for each component of the system developed here. These include

1. an user manual for BeeHive[6],
2. a systems document for the Behavior Engine[7], and
3. an article explaining the compiler[8].

More detailed information of the implementation of the system can be found in these documentation.

5.3 Rounding Up

In this thesis, we outlined the current state of events for honeypot deployment, and motivated the need for a behavior emulating system on honeypots. Such a system will have the honeypot perform actions consistent with that of a real production system, and enhance the chances of successful detection evasion.

As part of such a behavior emulating system, this thesis proposes a scripting language **BeeHive** that is used to specified desired behaviors to exhibit, alongside with a **Behavior Engine** that uses information scripted in BeeHive to determine the desired actions to perform on the honeypot. As an interface between the BeeHive language and the engine, a BeeHive **compiler** is also suggested to help facilitate the translation from BeeHive to the C++ source of the Behavior Engine.

We have shown that BeeHive is sufficiently expressive and that the Behavior Engine is an efficient platform with which to deliver the desired emulated behaviors. With continual advancement in the artificial intelligence fields of content generation, this platform can only improve and better deliver on its goal of enhancing the stealthiness of honeypots.

References

[1] Holz, Thorsten and Raynal, Frederic. *Detecting Honeypots and Other Suspicious Environments*. Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, June 2005.

[2] *honeyd*. `http://www.citi.umich.edu/u/provos/honeyd/`.

[3] Honeynet Project. `http://www.honeynet.org/index.html`.

[4] IEEE Standards. `http://standards.ieee.org/regauth/oui/oui.txt`.

[5] Intrusion Detection, Honeypots, and Incident Handling Resources. `http://www.honeypot.net`.

[6] Ng, Jun Ping and Cai, Alvin. *BeeHive — What will you do on a honeypot?*. 2006.

[7] Ng, Jun Ping and Cai, Alvin. *Behavior Engine — System Documentation*. 2006.

[8] Ng, Jun Ping. *From BeeHive to C++ — Exploring the BeeHive Compiler*. 2006.

[9] Oberheide, Jon and Karir, Manish. *Honeyd Detection via Packet Fragmentation*. Networking Research and Development, Merit Network Inc., 2006.

[10] Sebek. `http://www.honeynet.org/tools/sebek/`.

[11] Schneier, Bruce. *Secrecy, Security and Obscurity*. Crypto-Gram Newsletter, 2002.

[12] Snort. `http://www.snort.org`.

[13] Spitzner, Lance. *Honeypots: Definitions and Values*. `http://www.csd.uoc.gr/ gvasil/stuff/honeypots/honeypots.html`, 2003.

Appendix A

BeeHive Grammar

```
1
2    :: Grammar for BeeHive
3         v 1.0
4       Tue Mar 21 16:54:16 SGT 2006
5
6    :: Ng, Jun Ping - njunping@dso.org.sg
7
8
9  <construct> ::=
10       <action-construct>
11     | <event-construct>
12     | <time-statement>
13     | <action-time-logic>
14
15
16 <action-construct> ::=
17     CREATE ACTION <name> {
18       <set-task-line>
19       <set-attribute-lines>
20     }
21
22 <event-construct> ::=
23     CREATE EVENT <name> {
24       <set-attribute-lines>
25       <do-construct>
26     }
27
28 <set-task-line> ::=
29     SET TASK TO BE <string>
30
31 <set-attribute-lines> ::=
32       /* empty */
33     | <set-attribute-line>
```

```
34    | <set-attribute-line> <set-attribute-lines>
35
36  <set-attribute-line> ::=
37      SET ATTRIBUTE <name>
38    | SET ATTRIBUTE <name> TO BE <attribute-value>
39
40  <attribute-value> ::=
41        <number>
42    | <distribution-spec>
43
44  <distribution-spec> ::=
45      <name-of-distribution>(<distribution-param-list>)
46    | <name-of-distribution>()
47
48  <name-of-distribution> ::=
49    _UNIFORM_DISRIBUTION | _NORMAL_DISRIBUTION
50
51  <distribution-param-list> ::=
52        <number>
53      | <number>, <distribution-param-list>
54
55  <do-construct> ::=
56      /* empty */
57      | DO <do-settings> {
58        <do-body>
59        }
60
61  <do-settings> ::=
62        /* empty */
63      | WITH <do-setting-param> <number>
64      | WITH <do-setting-param> <number> AND
            <do-setting-param> <number>
65
66  <do-setting-param> ::=
67        urgency
68      | priority
69
70  <do-body> ::=
71      /* empty */
72    | <specifier> {
73        <do-body-parts>
74      }
75    | <repeat-specifier> {
76        <do-body-parts>
77      } <until-specifier> {
78        <ensure-body>
79      }
80
81  <do-body-parts> ::=
```

```
82      /* empty */
83    | <do-block> <do-body-parts>
84    | <do-statement> <do-body-parts>
85
86  <do-block> ::=
87      <percentage-spec> <do-body>
88
89  <do-statement> ::=
90      <percentage-spec> <name>(<do-param-list>)
91      | <percentage-spec> <name>(<do-param-list>)
92    | <percentage-spec> <name>()
93    | <percentage-spec> <name>()
94    | <percentage-spec> <interrupt-statement>
95    | <percentage-spec> <resume-statement>
96
97  <percentage-spec> ::=
98      WITH <decimal_number>% CHANCE
99    | WITH <decimal_number>% CHANCE AND <decimal_number>%
        INTERRUPTABILITY
100   | WITH <decimal_number>% INTERRUPTABILITY
101
102 <do-param-list> ::=
103     <do-param>
104   | <do-param>, <do-param-list>
105
106 <do-param> ::=
107   <name>
108
109 <specifier> ::=
110     SEQUENCE
111   | PARALLEL
112   | CHOOSE <number> OF
113
114 <repeat-specifier> ::=
115   REPEAT
116
117 <until-specifier> ::=
118   UNTIL
119
120 <interrupt-statement> ::=
121   INTERRUPT
122
123 <resume-statement> ::=
124   RESUME
125
126 <time-statement> ::=
127   SET TIME <name> TO BE <number>
128
129 <action-time-logic> ::=
```

```
130     <time-qualifier> TIME <name> {
131     <ensure-construct>
132     <do-construct>
133     }
134   | <time-qualifier> ACTION <name< {
135     <ensure-construct>
136     <do-construct>
137     }
138
139 <time-qualifier> ::=
140     AT
141   | BEFORE
142   | AFTER
143
144 <ensure-construct> ::=
145     /*empty */
146   | ENSURE {
147       <ensure-body>
148     }
149
150 <ensure-body> ::=
151     /* empty */
152   | <ensure-statement>
153   | <ensure-statement> <ensure-body>
154
155 <ensure-statement> ::=
156     <condition> <rel-operator> <boolean>
157   | <condition> <rel-operator> <number>
158   | <condition> <rel-operator> <name>
159
160 <condition> ::=
161     <name>.<action-status>
162   | <function-call>
163
164 <action-status> ::=
165     BEGUN
166   | DONE
167   | TIMES
168
169 <function-call> ::=
170     <name>()
171   | <name>(function-param-list)
172
173 <function-param-list> ::=
174     <function-param>
175   | <function-param>, <function-param-list>
176
177 <function-param> ::=
178   <name>
```

```
179
180 <rel-operator> ::=
181     ==
182   | !=
183   | <
184   | >
185   | <=
186   | >=
187
188 <boolean> ::=
189     TRUE
190   | FALSE
191
192 <decimal_number> ::=
193     <number>.<number>
194   | <number>
195
196 <number> ::=
197   <digit>+
198
199 <digit> ::=
200   [0-9]
201
202 <string> ::=
203   "<name>"
204
205 <name> ::=
206   [a-z][a-zA-Z0-9_]*
```