

VICTORIA UNIVERSITY OF WELLINGTON  
*Te Whare Wānanga o te Ūpoko o te Ika a Māui*



School of Engineering and Computer Science  
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600  
Wellington  
New Zealand

Tel: +64 4 463 5341  
Fax: +64 4 463 5045  
Internet: [office@ecs.vuw.ac.nz](mailto:office@ecs.vuw.ac.nz)

**Zombie Beatdown: Automating the  
Discovery of Web Malware**

Micah Cinco

Supervisors: Ian Welch and Masood Mansoori

Submitted in partial fulfilment of the requirements for  
Bachelor of Engineering (Hons).

**Abstract**

With a significant increase in user population of various web applications, drive-by download attacks (DBD) has become one of the major security threats on the web infrastructure. With the dangers that it imposes on vulnerable web browsers, the improvement and enhancement of honeypots have seen a significant growth in recent years. Capture-HPC, a system that is able to check the system and detect malware by starting and terminating processes, checking file system modifications and checking registry modifications, has presented limitations to its system in terms of compatibility with newer versions of Windows operating systems. Cuckoo, an open source project, does not have such limitations but do not have the same management function that Capture-HPC does, thus, limiting its analysis process to just one web server at a time. The project looks at how the Cuckoo Sandbox system can be used to automate the discovery of web malware and how a system manager can be implemented to create and manage multiple instances of the Cuckoo client. The literature review conducted looks at existing architectures similar to that of Cuckoo and compares the advantages and disadvantages of each system with the proposed solution. The system was tested and a performance overhead, along with correctness of implementation through the tracking task flows was evaluated.



# Acknowledgments

I would like to thank my supervisors, Dr Ian Welch and Masood Mansoori. Dr Ian Welch gave me the opportunity to be part of this project which allowed me to gain knowledge and insight on web malware, security and honeypots. Personally, I found all knowledge gained throughout the project to be very valuable, being someone who wants to specialize in that field of Network Engineering. He has also provided invaluable support as a mentor and supervisor, providing guidance throughout my university studies, especially on my final year.

Masood Mansoori, who I've had the pleasure of knowing this year, has been incredibly helpful and supportive in the initial stages of the project, helping me understand what Cuckoo is actually all about and provided valuable feedback whenever possible.

To the staff in the Network Engineering faculty, especially to the ones that I've had as lecturers this year. For encouraging me throughout the year, being understanding and providing advice at one point or another.

To my family, for being incredibly supportive, not just this year but throughout my university studies, motivating me and encouraging me to keep going even on the craziest times of the year.

Lastly, to my peers. You have made my final year of university enjoyable and your constant help and support, even on the smallest things, have been invaluable to me.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project Goals . . . . .	1
1.2	Report Organization . . . . .	2
<b>2</b>	<b>Background Review</b>	<b>3</b>
2.1	Drive-by Downloads . . . . .	3
2.1.1	Virtual Environments . . . . .	3
2.2	Detection of Drive-by Downloads . . . . .	4
2.2.1	Capture-HPC . . . . .	4
2.2.2	HoneySpider Network 2.0 . . . . .	5
2.2.3	Puppet . . . . .	6
2.2.4	Cuckoo Sandbox . . . . .	6
2.3	System Architecture . . . . .	8
2.3.1	Client-Server . . . . .	8
2.3.2	Master-Slave . . . . .	8
2.4	Fault Detection . . . . .	9
2.4.1	Timeouts . . . . .	10
2.4.2	Heartbeats . . . . .	10
<b>3</b>	<b>Design</b>	<b>11</b>
3.1	Terminology . . . . .	11
3.2	Design Requirements . . . . .	11
3.3	Project Assumptions . . . . .	12
3.4	Design Decisions . . . . .	13
3.4.1	System Architecture . . . . .	13
3.4.2	Fault Detection . . . . .	13
3.4.3	Storage . . . . .	14
3.4.4	User Interface . . . . .	14
3.5	Final Design . . . . .	14
3.5.1	Architecture Overview . . . . .	14
3.5.2	Cuckoo Manager . . . . .	17
3.5.3	System Analysis . . . . .	19
3.5.4	Satisfying Design Requirements . . . . .	20
<b>4</b>	<b>Implementation</b>	<b>21</b>
4.1	Development Environment . . . . .	21
4.1.1	Python . . . . .	21
4.1.2	Eclipse IDE and Python Libraries . . . . .	21
4.1.3	VirtualBox . . . . .	22
4.1.4	MongoHQ . . . . .	22

4.1.5	GitHub . . . . .	22
4.2	Zombie Beatdown Manager . . . . .	22
4.2.1	Scripts . . . . .	23
4.2.2	GUI . . . . .	25
4.3	Zombie Beatdown Worker Analysis . . . . .	26
4.3.1	Scripts . . . . .	26
4.3.2	User Interface . . . . .	27
<b>5</b>	<b>Evaluation</b>	<b>28</b>
5.1	Evaluation Environment . . . . .	28
5.2	Functional Testing . . . . .	28
5.2.1	Zombie Beatdown Manager - Task Management . . . . .	29
5.2.2	Zombie Beatdown Worker - Task Analysis . . . . .	31
5.3	Performance Testing . . . . .	34
5.3.1	Network Latency . . . . .	34
5.3.2	Average Job Time . . . . .	35
5.4	Limitations of Zombie Beatdown . . . . .	37
5.5	Evaluation Summary . . . . .	37
<b>6</b>	<b>Conclusion and Future Work</b>	<b>38</b>
6.1	Project Contributions . . . . .	38
6.2	Future Work . . . . .	39
6.2.1	Improved Fault Detection Scheme . . . . .	39
6.2.2	Result Reporting Visualization . . . . .	39
6.2.3	Analysis of Task Results . . . . .	39

# Figures

2.1	Cuckoo's Main Architecture . . . . .	7
3.1	Flow of execution for Zombie Beatdown . . . . .	15
3.2	Sequence Diagram for Creating a Task . . . . .	16
3.3	Sequence Diagram for Checking Tasks . . . . .	16
3.4	Sequence Diagram for Creating a Task by Progress . . . . .	17
3.5	Sequence Diagram for Creating a Task by ObjectID . . . . .	17
3.6	Wireframes of the three web pages for the Zombie Beatdown Manager . . . .	18
3.7	Sequence Diagram for Pulling a Task . . . . .	20
4.1	RESTful Service API for Manager's web server . . . . .	23
4.2	Handler for POST request /send . . . . .	23
4.3	"Delete by Progress" script in mongo.py . . . . .	24
4.4	"Create a Task" web page . . . . .	25
4.5	HTML form code for "Check a Task" web page . . . . .	25
5.1	Message returned after successful task creation . . . . .	29
5.2	Database after successful task creation . . . . .	29
5.3	Tasks returned when checking for "idle" tasks . . . . .	30
5.4	Manager GUI when deleting task by ObjectId . . . . .	31
5.5	Pulling of Tasks with "High" priority . . . . .	32
5.6	2nd Pull for Task with "High" priority . . . . .	32
5.7	Message returned when no tasks are in "idle" . . . . .	33
5.8	Successful run for a task . . . . .	33
5.9	Plot of Latency for "Pull Task" function with Varying URL Size . . . . .	34
5.10	Plot of Average Job Time for Varying URL Size . . . . .	36

# Chapter 1

## Introduction

With the rise of drive-by downloads, the risk of turning your computer into zombies controlled by malicious attackers is increased. These programs are usually downloaded or installed on computers without the owner's knowledge just through visiting a malicious website using a vulnerable browser. Honeypots are traps which are basically used to detect, deflects and sometimes even counteract attempts at unauthorized use of information systems. For my particular scenario, a client honeypot is a technology that allows one to find malicious servers on a network by actively searching for those that attack web browsers or any other application that access the underlying operating system.

Currently, there is an existing honeypot called Capture-HPC that is able to check the system and detect malware by starting and terminating processes, checking file system modifications and checking registry modifications. Capture-HPC was developed by students at Victoria University between 2006 and 2008 and was released as an open source software. It has been used by CGRTs in the Netherlands and in Poland as well as integrated into software produced by the MITRE corporation. Capture-HPC was limited to older versions of Windows operating systems so we are redeveloping Capture-HPC around another open source project called Cuckoo.

Cuckoo is an automated malware analysis sandbox. The overall aim of this project has been to add Capture-HPC's management functionality to Cuckoo in order to allow us to scale up and automate the analysis of multiple web servers. Over the summer, two interns from Singapore Polytechnic worked with PhD candidate, Masood Mansoori, on modifying Cuckoo to support the type of analysis that is similar to that of Capture-HPC but taking advantage of the extended functionalities that Cuckoo offer.

### 1.1 Project Goals

The goals of this project are as follows:

- Investigate various architectures and designs for management systems to establish an idea of how Cuckoo's proposed extension could be designed.
- Establish a set of requirements based on the literature review that the design specifications must meet.

- Design a management system for Cuckoo that would allow a user to push and manage tasks of multiple URLs to a database then pull these tasks, locally or from a remote location, to enable an automated analysis of multiple URLs.
- Implement and evaluate the management system, assessing its effect on overall job time to process multiple URLs, as well as ensure normal functionality of the existing Cuckoo implementation is not altered.

## **1.2 Report Organization**

The rest of this report is structured as follows - In Chapter 2, a background review is performed which includes a literature review of existing management systems, different architectures applicable to such systems and an analysis of fault detection approaches. Chapter 3 covers the design for the project, discussing the constraints, design requirements, approaches considered for the project and the final overall architecture chosen. Chapter 4 gives insight on the implementation that went into the project, detailing the different scripts and extensions made to Cuckoo. Chapter 5 presents the evaluation for Zombie Beatdown, showing functional and performance test results, followed by Chapter 6 which gives concluding remarks of the project and discusses opportunities of future work.



## Chapter 2

# Background Review

### 2.1 Drive-by Downloads

With a significant increase in user population of various web applications, drive-by download attacks (DBD) has become one of the major security threats on the web infrastructure. A user who may be using a vulnerable web browser or plugin can be attacked by visiting a malicious website that can inject malicious scripts using Javascript or PHP or through HTML tags to force users to download malicious software. It was found that more than 1.2% of query results provided by Google redirects to a web server that performs drive-by download attacks. Furthermore, these compromised websites were found to have very short time spans and come and go so often that it becomes hard to catch them, thus, prevention and defense against such attacks have become the main focus in the field.

Preventing users from accessing malicious web pages can become difficult due to the fact that there are many ways of redirecting users to visit a malicious website, therefore, works such as Google Safe Browsing have been implemented to filter and check URLs against a blacklist of malicious URLs on the user side. However, this approach is hard to be effective due to the ephemeral nature of malicious websites. Further work in this field is to attempt and keep the blacklists up to date. Another approach taken against drive-by downloads are the use of high interactive web crawlers, called honeypots.

In this approach, honeypots are a form of Intrusion Detection Systems (IDS) which attempts to "identify, preferably in real time, unauthorized use, misuse, and abuse of computer systems by both system insiders and external menetrators" [5]. A honeypot is described in [2] as a virtual computer system which "attract hackers to fall into it to watch and follow their behaviour". Such systems try to detect intruders by mirroring the characteristics of a real system record all interactions and actions that a website visit does to the system.

#### 2.1.1 Virtual Environments

Virtual environments are a very useful tool in detection of drive-by download attacks since it allows the imitation of a real system but the ability to be flexible. The deployment of honeypots in virtual environments gives such a system the ability to run multiple operating

systems on one computer, reducing costs on purchasing hardware. Deployment of physical honeypots in real, physical systems are often expensive since each honeypot requires its own machine. Virtualization cuts the cost for this and allows flexibility with having more than one honeypot per physical machine. In addition, virtualization software such as VirtualBox, KVM and VMWare, provide users the ability to "roll back" or reset the system to a previous state through the use of a "snapshot", undoing any changes that may have been made by the malware.

## 2.2 Detection of Drive-by Downloads

With the huge increase in the number of malware samples, in recent years there have been several approaches to the investigation of automated malware analysis techniques by using management systems to create multiple instances of honeypots or sandboxes in virtual environments. These approaches range from high interaction client honeypots such as Capture-HPC, a scalable system to integrate multiple client honeypots like HoneySpider and Puppet, an open-source utility used to manage the configuration of systems declaratively [?, 11, 12, 13, 14].

The following section looks at each of these approaches and discusses the differences in strengths and weaknesses to investigate existing software architecture that needs to be considered for the project's design.

### 2.2.1 Capture-HPC

The Capture-Honeypot Client (Capture-HPC) is a high interaction Honeypot that actively searches for malicious websites by interacting with potential malicious servers and using a dedicated virtual machine and observing the system for unauthorized state changes. If the system detects such changes, that website is then classified as 'malicious', and all events are logged by the Capture-Server before resetting the virtual environment to its clean state using a snapshot, ready for the next website. Hes et al [13] outlines that communication protocol of the Capture-HPC architecture and illustrates how the system's design, made of two parts Capture-Client and Capture-Server interacts with each other during execution to detect the presence of malicious servers.

Typically, the server is run on one host machine and the clients are run on their own virtual environments, either on the same or separate physical host. The two communicates to each other by passing XML messages over a simple TCP socket connection, established by the client, to allow the server to issue a batch of instructions to the client which in turn, directs it to a set of applications to visit the remote material [14].

When visiting applications, Capture-HPC may be run in three different modes sequential, bulk or divide and conquer. In sequential mode, the Capture-client visits the set of URLs sequentially whereas bulk mode executes a set of visits all at once. In divide and conquer mode, the Capture-client visits the websites as a bulk but would perform a divide and conquer algorithm to isolate the offending site upon detection of a malicious network.

One of the important things that the Capture-Server should also be able to do is detect

when a particular Capture-client crashed or fails to execute a task. To do so, Capture-HPC's socket-based nature allows the communication protocol to contain a heartbeat mechanism that determines whether the client is still alive and responsive to Capture-Server commands [14]. The heartbeat mechanism utilizes the same structure as XML over TCP/IP but is run on a different thread to make it independent of Capture-client/server activity. Heartbeat works by the Capture-Server pinging the Capture-Client to solicit a heartbeat response. The Client then responds by sending ICMP packets back to the Server to acknowledge the message and let it know that it's still alive and responsive. If the heartbeat stops, the Capture-Server will assume that the Client is no longer alive and will initiate a reset of the virtual machine in order to recover from the error and subsequently allow the Client to attempt a new connection with the Server.

### 2.2.2 HoneySpider Network 2.0

The HoneySpider Network (HSN) 2.0 is a highly-scalable system that integrates multiple honeypots, dedicated to detect the presence of malicious websites in the Internet. The system focuses primarily on attacks involving the use of web browsers and allows detection of drive-by downloads and malicious binaries. Originally, the HSN was managed by a centralized manager that composed of a low interaction manager and a high interaction manager that manages its respective honeyclients.

The new release for HSN consists of the system's main component, the Framework, which facilitates the information being exchanged between different components. The HoneySpider Network uses open-source tools available to process data, like HtmlUnit and RabbitMQ and a mixture of low and high interaction honeyclients such as Thug, Cuckoo Sandbox, Capture-HPC and a static JavaScript analyzer to analyze data. The whole system can be used directly from a web browser or the command line interface.

HoneySpider Network uses the Advanced Message Queueing Protocol (AMQP), a message-oriented protocol, to communicate between components as well as some use of the RESTful API. CouchDB is utilized for storage of results in JSON format and the use of operational data and flexible mapping allows for persistent reports.

The architecture for HSN comprises of five main components Import Layer, Filter Layer, Analysis Layer, Management Layer and the Presentation Layer [12]. The Import Layer is responsible for the initial acquisition of URLs that are to be processed by the system. The Filter Layer involves the maintenance of white, grey and black lists. The whitelist consists of URLs that have been deemed benign, blacklist has the URLs that are malicious and greylist contains the URLs that have been identified as suspicious. For all three lists, URLs won't be checked until after a specified period of time that can be explicitly defined or in the form of a Time-to-Live value (TTL). The Analysis Layer is where the low and high interaction honeypots operate and is responsible for the detection of malicious websites, exploits used and malware downloaded. Both low interaction and high interaction components have their local processing queues, maintained by their local managers. Once a URL is processed, it is submitted back to the central processing queue, where further decisions are made as to how a URL should be processed [?]. The Management Layer controls the flow of objects within the system, maintains the configuration, controls import processes, management of queues, scheduling and allocation of objects to low and high interaction processing units. Each object is also given a final alert classification, *NOT\_ACTIVE*, *BENIGN*, *SUSPICIOUS* or

*MALICIOUS*, after being processed and is tagged with other information on its confidence level, priority level and process classification. Finally, the Presentation Layer allows the user to access and view all information concerning processed objects through a web-based GUI. An alerter plug-in and reporter plug-in is also available to either send alerts via email or SMS or create PDF reports with the graphical statistics and/or detailed information, respectively.

### 2.2.3 Puppet

Unlike Capture-HPC and HoneySpider Network 2.0, Puppet is not dedicated to security-related issues such as analysis or discovery of web malware. Instead, it is an open-source utility used to manage the configuration of two or more systems declaratively and is mainly used for datacenter automation and server management. When Puppet is configured, it automatically updates all machines to match the configuration specified in the controller. Puppet runs in a client/server mode with a central server and agents running on separate hosts. For this literature review, we only investigated the network communication protocol for Puppet and a high level view of its components.

Puppet works through the principle of least privileged in that each client only knows what it needs to know which is how it should be configured. It does not know how the servers are configured and gets the configuration file compiled by the server and passed on to the client. The system also allows clients to run whilst disconnected from the central server by repeatedly applying the configuration and provides functionality even if it is disconnected or if the central server is down. Furthermore, the responsibility of compiling a configuration and applying the configuration is explicitly differentiated strictly defines access permissions to central data stores. This approach loosely couples the functionality between client and server and makes the system more scalable in that there is no need to update the server or the whole system when updating one client. The Puppet agent first collects information about the host that it is running on and passes this to the server which then uses that data to compile a configuration for that host and returns it back to the agent. The agent then applies the configuration locally and saves a resulting report in the server after completion.

In terms of networking and its communication protocol, Puppet originally used XML-RPC which caused memory problems due to the protocol's encoding's need to save every object twice [11]. Because of this, it was decided to, instead, use a REST-like model and switch over to using JSON for serialization rather than YAML. YAML processing in Ruby, which is the language used implement Puppet, became painfully slow and processing JSON in Ruby was found to be more efficient and compatible with majority of other programming languages.

### 2.2.4 Cuckoo Sandbox

Cuckoo Sandbox is a malware analysis system allows a user to analyze a file which returns a detailed report outlining what that file executed inside an isolated Windows operating system. Currently, Cuckoo gives the user the capability of running one instance of the sandbox to analyze a given file URL, dll, xls, doc, pdf, jar, exe, zip, etc. Cuckoo then generates a report in JSON, HTML, MAEC, or in MongoDB or HPFeeds interfaces that include native functions and Windows API call traces, copies of files created and deleted from the file

system, dump of memory for selected processes, full memory dump of the analysis machine, screenshots of the desktop during execution of the malware analysis and network dump generated by the machine used for the analysis. Unlike Capture-HPC, which has a server-side manager that handles multiple instances of the client and determines from the report whether the visited website is malicious or not, Cuckoo is limited to just one instance running at a time and generates a report back to the user.

Written in Python, Cuckoo consists of a central management software that handles file execution and analysis. Each analysis is launched in an available virtual environment which is the clean copy of a "snapshot". Cuckoo's whole infrastructure is hosted in a physical machine, called the Host machine, which runs the management software and coordinates one or more Guest machines, which are the virtual machines used for analysis. When a Host runs the core component of Cuckoo, it manages the whole analysis process whilst the Guests run the malware or visit the web servers and perform analysis in a safe, and isolated environment, not allowing the physical system to be affected by any malicious activity.

Figure 2.1 illustrates Cuckoo's main architecture.

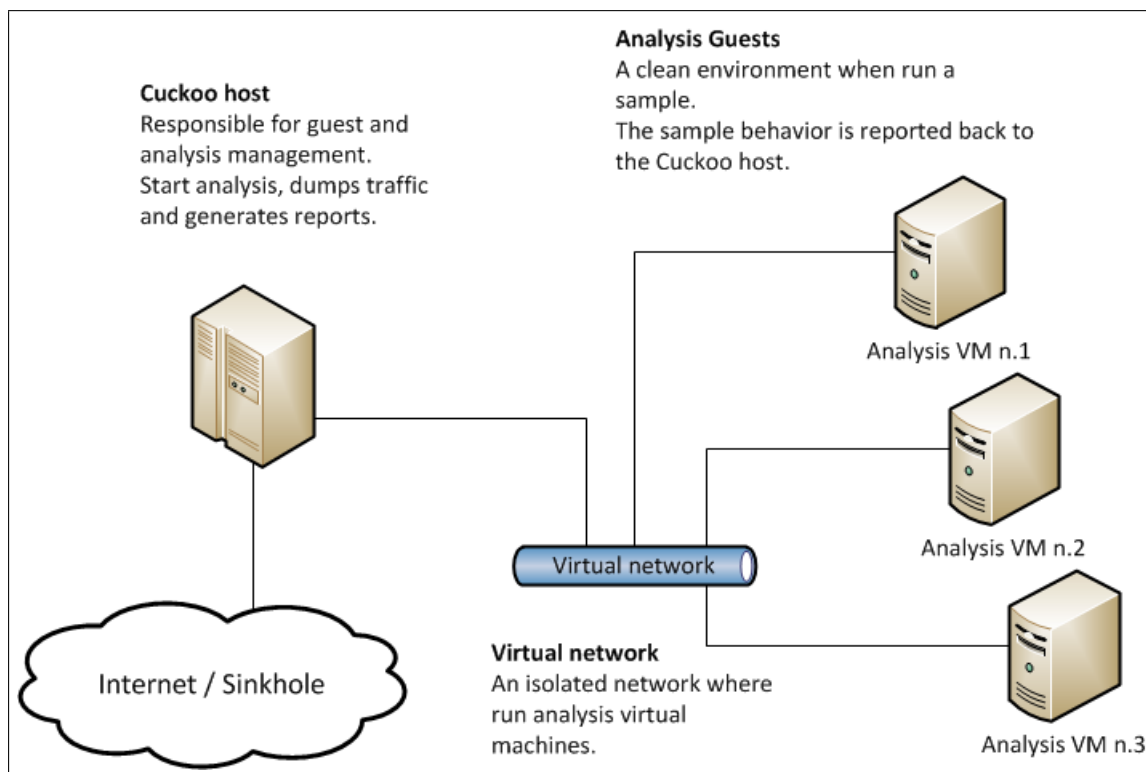


Figure 2.1: Cuckoo's Main Architecture

Similar to Capture HPC's design, Cuckoo utilizes a Master-Slave architecture as discussed in Section 2.3.3 of this report. With a Master-Slave approach, the Cuckoo Host is able to collect a set of files or websites to analyze which gets pushed out to available virtual machines and all analysis is performed by the Guest machines. This centralized model allows the master or Host to focus on managing tasks whilst slaves or Guests have one task of analyzing the sample and sending back a detailed report.

Cuckoo's growing popularity has also established a very strong online community which

provides extensive support and examples on how to extend Cuckoo's functionalities which make it a flexible system to use since the user is able to customize it to their needs.

## 2.3 System Architecture

The following section looks at two of the many architectures for distributed computing, Client-Server model and Master-Slave model. Here, we investigate the overall architecture's model, as well as looking at the advantages and disadvantages of each design.

### 2.3.1 Client-Server

Client-Server architecture is a computing model where server hosts, delivers resources to a client. In such a model, one or more client machines are connected to a central server over a network through the Internet. In [?], Hura discusses the Client-Server computing model and how it utilizes resources efficiently and effectively. In such systems, when a client wants to retrieve a file from a server, it can then send a request to that server, which in turn sends the entire file back to the client. If another user wants to retrieve that exact same file, for example, train schedules, then the whole process is repeated. With this model, communication costs are more expensive due to a higher network traffic but the power usage with both client and server are less expensive. This architecture also allow the server to provide resources to multiple clients simultaneously, whilst clients can also be connected to multiple servers at a time.

In terms of security, since the server has full access on who it can share its data or resources with, they also have better control over access control and ensuring that only authorized clients can access or manipulate data. Furthermore, this model improves data sharing between the server and clients since it supports open access to its users with different operating systems. With the development of Content Distribution Networks (CDN), servers are now also geographically distributed to serve clients within its proximity, thus, improving its service to client machines that may be located in various locations. The impact of a centralized architecture, however, is the load that it imposes on the server. With many requests coming in from various clients, servers may get severely overloaded, forming traffic congestion. Furthermore, this also means that the bottleneck is on the central server, thus, if a server fails, then client requests are not accomplished unless a backup server is available.

### 2.3.2 Master-Slave

In [6], the authors discuss different design patterns for software architecture and one that is relevant to the project is the Master-Slave design pattern. This architecture supports fault tolerance, parallel computation and computational accuracy and involves a master component distributing the workload to identical slave components to compile the final results that these slaves return. This approach takes on the 'divide and conquer' principle which allows partitioning of work into several subtasks that can be processed independently to allow the master component to compile the results and deduce a final result from that. Furthermore, the Master-Slave design provides all slaves with a common interface and allows them to

communicate only with the master. The slave has a subservice that processes the sub-tasks allocated to it by the master. In a typical scenario, an external client would request a service from the master, which in turn partitions that task into several sub-tasks and delegates the execution to several slaves. It then starts the processes and waits for the results from all slaves. The slaves complete execution and return their results back to the master to compute the final result for the whole task before sending it back to the client.

One of the areas that the Master-Slave pattern is utilized for is fault tolerance where as long as at least one slave does not fail, then the client is provided with a valid result. Typically, the master runs timeouts to detect whether a slave failure has occurred. If so, then the master would detect it and be able to perform a set of tasks to address the issue (e.g. refresh the slave and reissue the sub-task). The disadvantage of a Master-Slave paradigm however is the importance of having the master component alive at all times as otherwise, the whole structure would not work. Another area that this architecture is used for is parallel computation since all sub-tasks are being executed simultaneously between slaves. This aspect of the pattern is strongly dependent on the hardware architecture and speed of processors of the machine with which the program runs and other aspects such as the existence of shared or distributed memory for machines. To free the master from attempting to synchronize its slaves individually, [6] presents a concept, barrier. A barrier is basically initialized on slaves to suspend the master from running and collecting the data until all sub-tasks allocated have been executed completely. Master-Slave is also used for computational accuracy where at least three implementations are used for sub-task execution. The master then waits for all processes to execute then uses a number of different strategies for the algorithm that collects the data. The master can either select the result returned by the greatest number of slaves or the average of all results. To allow different slave implementation, it is also possible to extend the structure by adding an abstract class that acts as a middle man between the slaves and the master.

By introducing an abstract slave class, it allows the system to be extensible and have the possibility of exchanging existing slave implementations or add new ones without major changes to the master. Also, if implemented carefully, performance and execution of particular services can be increased but would affect the cost of parallel computation. This software pattern also has its drawbacks and having a system that partitions work, copies data, launches slaves, control execution and compile results make it less likely to be feasible due to these activities consuming processing time and storage space. This also means that careful consideration must be done when implementing this pattern as it would not be easy. It must have the ability to deal with errors such as failure of slave execution, failure of communication between master and slaves or failure to launch a particular parallel slave.

## 2.4 Fault Detection

Failure detection and recovery are main components to fault tolerant computing in distributed systems. These systems must be able to handle crash failures such as a crash with the communication between the slave and master. In distributed systems, a common way to handle crashes involves two steps: (1) Detect the failure and (2) Recover by restarting the crashed machine.

### **2.4.1 Timeouts**

Current approaches to failure detection involve the use of an end-to-end timeout that is set for a particular set of seconds. This approach, however, means that if a machine crashes, the system may be unavailable for a long period of time waiting for the timer to timeout [16]. An obvious alternative would be to set the timeout to a shorter span of time but this would increase the risk of declaring that a fully functional working node, with a possible slow network connectivity, has crashed.

### **2.4.2 Heartbeats**

[16] discusses another way for failure detection which is use of a time-out free concept called, heartbeat. With heartbeat-based failure detectors, the basic idea is that each process has attached to it a failure detector module that continuously outputs an estimate of which processes in the system have failed. A heartbeat scheme can work well if the heartbeat message is delivered to the failure detection module within a fixed period of time. Yang et al. [16] presented a heartbeat-based scheme that takes on a notification-based approach and lets the host send notification messages to interested entities once a process has failed. This approach increases efficiency as the message cost is lower than that of a nave approach where the host is constantly pinging the relevant machines until it is no longer receiving an ACK. The suggested approach is also scalable where it increases the message cost linearly with the number of process failures, no matter how many processes or machines are involved and accurate the output does not oscillate between suspicious processes to trusted ones. Lastly, heartbeat-based systems are also simple and easy to implement since there are no need for synchronized clocks.



# Chapter 3

## Design

This chapter presents the design decisions and overall layout of Zombie Beatdown that are used to implement the server-side manager and extension of the existing Cuckoo worker. First, the project constraints are presented by discussing the design requirements and assumptions which have been extracted from our findings in the literature review. The design decisions for Zombie Beatdown are then discussed, addressing various operations that could be applied and justifies why each direction was chosen. The last section goes through the final design of Zombie Beatdown, outlining the classes that will be implemented, the manager's user interface, and provides an overall layout showing the complete execution flow of the system.

### 3.1 Terminology

Before proceeding to the discussion of my final design, I'd like to establish some terminology used for the main components of the system:

- *Manager* - the web server tool in charge of creating, checking and deleting tasks.
- *Database* - the MongoDB, noSQL database hosted in the cloud platform, MongoHQ, and stores all tasks created by the Manager.
- *Worker* - the physical machine that hosts the virtual machines used to execute task analysis. This machine is what pulls a task from the database, manages the execution of URLs within a task and is the machine that a user interacts directly with.
- *Minion* - the virtual machine that performs the actual URL analysis and is assigned, by the worker, one URL to analyse at a time.

### 3.2 Design Requirements

Before any design decisions are made, it is important to outline the set of requirements needed for Zombie Beatdown. These requirements have been extracted from the observa-

tions of the overall project goals (Section 1.1), and observations made from the literature review (Chapter 2).

The following outlines the requirements that should be met in the systems design:

1. Any existing functionalities with the Cuckoo sandbox should function as expected. The additional functionality of being able to process multiple URLs by pulling tasks from the database, should not affect the existing system as we do not want to inconvenience users who want to use Cuckoo in its original state. Such a scenario may then lead to reluctance in using the implemented functionality and make it harder for users to automate the discovery of web malware by having to individually process URLs.
2. Multiple workers can be geographically distributed and as long as workers have access to the internet, they should easily be able to pull a task from the database and execute `worker.py` to process the URLs.
3. Cuckoo minions, which are the virtual machines within a particular worker machine, may come and go without requiring changes made in the server-side.
4. The worker should pull tasks based on priority, thus, providing a load balancing mechanism within the system that allows higher priority tasks to be executed before lower priority tasks.
5. The manager can put tasks in the database, check the status of tasks and delete tasks from the database.
6. The Zombie Beatdown manager should be able to operate in the same manner with or without any workers active. This allows a fully decoupled system and enables the master to still operate as per usual without being dependent on active workers.

### 3.3 Project Assumptions

The following outlines the assumptions made for this project:

1. The database will be hosted using MongoHQ and is assumed to always be available and online.
2. We assume that the database has enough memory to store the tasks.
3. The Cuckoo worker, which hosts the minions (virtual machines) is already set up and ready to process files.
4. We assume that the users for the system have experience or knowledge on running the Cuckoo sandbox and the fundamentals of how it is operated.
5. Storage space on the Cuckoo host can be expanded and storage efficiency is not within the scope of the project.
6. Performance is dependent on the Cuckoo host's processors. If scalability is compromised due to hardware, host is able to upgrade.

## 3.4 Design Decisions

Creating the extension for Cuckoo to automate the processing of multiple URLs under one task required careful planning and thought, thus, requiring comparison of many differing approaches. Based on the literature review in Chapter 2, this section looks at the various design options that came up when creating Zombie Beatdown along with justifications on why each design decision was made.

### 3.4.1 System Architecture

After investigating the architecture for each system discussed in the literature review, I am able to produce a design, based on the requirements outlined above, that is similar to that of Capture-HPC's but takes advantage of a loosely coupled system by having the manager and database or the worker and database to communicate via HTTP using a RESTful approach, like Puppet.

The use of a master-slave system allows workers to perform and complete tasks and the master to still gain a valid result if workers fail or become unavailable, by having the tasks executed by other available workers. However, unlike a typical master-slave system where the master must always be active in order to track tasks, my design utilizes a database which stores the tasks that are pushed by the master, and allows available workers to pull and update tasks in the database. With this slight extension to the typical master-slave design, the system is still able to operate with or without an active manager as long as the database is always active (as outlined in Project Assumptions).

### 3.4.2 Fault Detection

For the fault detection mechanism of the system, two mechanisms were considered. The first fault detection mechanism considered is a combination of both end-to-end timeouts and use of heartbeats. As described earlier, a timeout will commence once a worker pulls a task from the database and if the worker completes execution within that timeout period, then the worker changes that task's state to completed, then updating the task in the database. If the worker does not complete execution by the end of the timeout, then the Taskhandler, which keeps an account of all active tasks and their timeout, informs the manager about it which in turn, pings the minion directly as in a heartbeat-based scheme. If the minion does not respond with an ACK then it is deemed no longer active and the Taskhandler initiates a restart for the virtual machine to recover from the error. Furthermore, rather than reissuing the task in again, the manager will simply change the task's current state from, "inprogress", back to "idle".

The second fault detection design makes use of the existing fault detection mechanism with Cuckoo where a URL is submitted for analysis and sets a countdown for that task before being assigned to an available minion. If the countdown times out, a task is reissued to another available minion and the timeout is refreshed. If no other minions are available, then the worker would be required to restart the whole task as "idle", abandon all completed analysis as part of the task before updating the database. If another worker becomes available and pulls that reissued task, then the whole cycle starts again. With this design,

the fault detection mechanism is less dependent on an external taskhandler and is managed by a set of rules that workers must adhere to.

### **3.4.3 Storage**

Whilst investigating the storage options for the manager, it was found that noSQL solutions like MongoDB was already supported by Cuckoo. Other databases such as CouchDB, Redis and Cassandra were also considered but since MongoDB was integrated with Cuckoo, it was easier to use existing technology built within the system. The decision to use non relational databases (noSQL) was due to their ability to handle unstructured data such as Word documents, emails, and multimedia files efficiently [10].

### **3.4.4 User Interface**

When looking at user interfaces, two possibilities would be to design a command line interface (CLI) or a graphical user interface (GUI). One of the project requirements outlines that all existing functionalities for Cuckoo must remain the same so in order to satisfy this requirement, having a command line interface for the worker is more appropriate since it is the existing user interface for Cuckoo and provides consistency within the system. Furthermore, since we're assuming that the target user has knowledge on how to operate Cuckoo, it is better to pull and analyse a task in the same manner as individually submitting a URL for analysis as this is the interface to which the user is accustomed to.

For the Zombie Beatdown manager, I decided to implement a GUI when creating, checking and deleting tasks since it would be easier to input and view information if it is in a point and click environment, to which most basic users would be familiar with, more so than a terminal environment. Although implementing a CLI is much quicker and easier to handle for the manager, it was more important to provide ease of use for the user and make it easier for anyone, with access to the database, to create, monitor and update tasks.

## **3.5 Final Design**

The final design for Zombie Beatdown is divided into two main components - the management of tasks and the execution of task analysis by the worker. This section presents a high level overview of Zombie Beatdown's architecture, followed by details of each component's functionality. The methodology of the whole system's workflow is then presented, giving insight to the scripts that is implemented and the flow of events at different stages of the execution.

### **3.5.1 Architecture Overview**

First, we present a complete overview of Zombie Beatdown. Figure 3.1 gives a visual illustration of the sequence that occurs within the main components in the system and the flow

of events from task creation, to workers executing the analysis of a task. The step-by-step process is as follows:

1. The manager creates a task using the GUI - providing the URLs that need to be analysed, the priority of that task and a specified timeout - then pushes the task to the MongoHQ database.
2. An active worker would pull a task from the database that is marked, "idle" and has the highest priority. Consequently, that worker will also change the task's current state to "inprogress".
3. The worker then parses the URLs in that task and assigns one URL per active minion.
4. Once all URLs in a task is completed, the worker updates the task in the database as "completed".
5. Finally, the manager can then check the database for task statuses and delete tasks either by their task ID or status (idle or completed). Tasks which are "inprogress" are not allowed to be deleted by the manager, thus the function is not provided, to avoid the deletion of a task that is being executed by a worker.

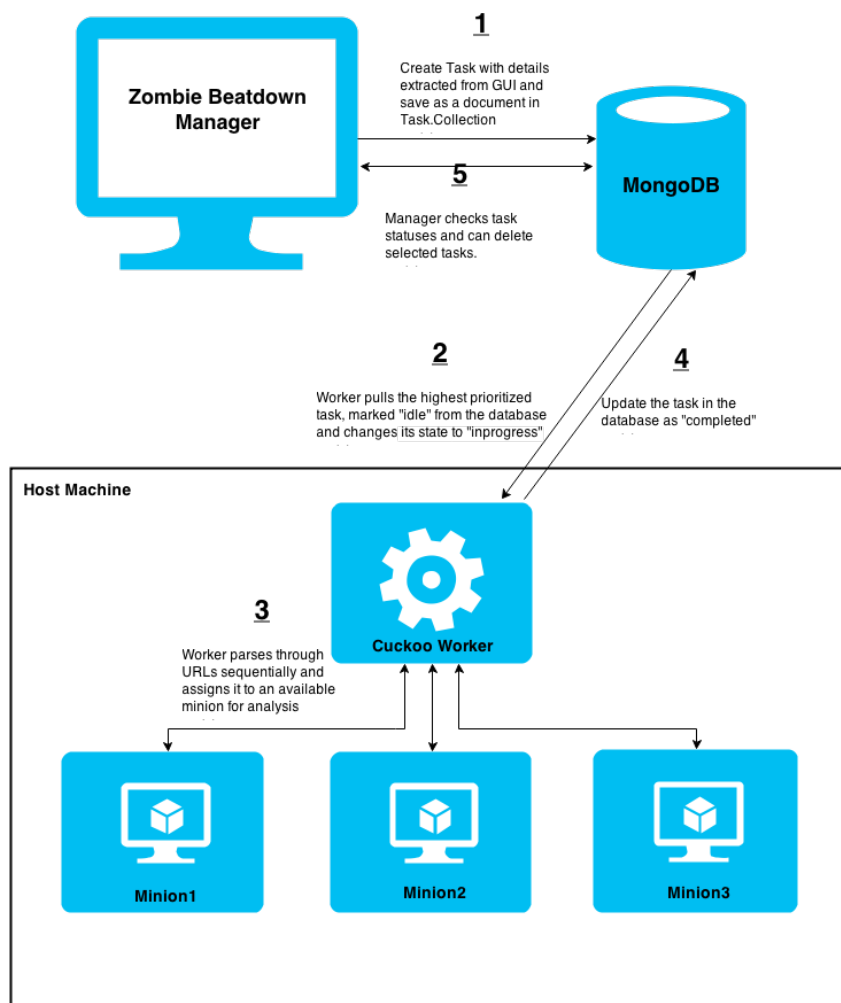


Figure 3.1: Flow of execution for Zombie Beatdown

As mentioned, there are several functions that the Manager may perform - creating a task, checking its status and deleting tasks from the database. Figures 3.2, 3.3, 3.4 and 3.5 present the sequence diagrams for each function and illustrates the communication for each scenario.

To satisfy the fifth design requirement outlined in section 3.2, we wanted to give the user the capability of creating, checking and deleting tasks from a GUI that is straight forward and simple. This interface will be the main point of interaction on the Manager-side of the system and the communication for each function is briefly outlined below.

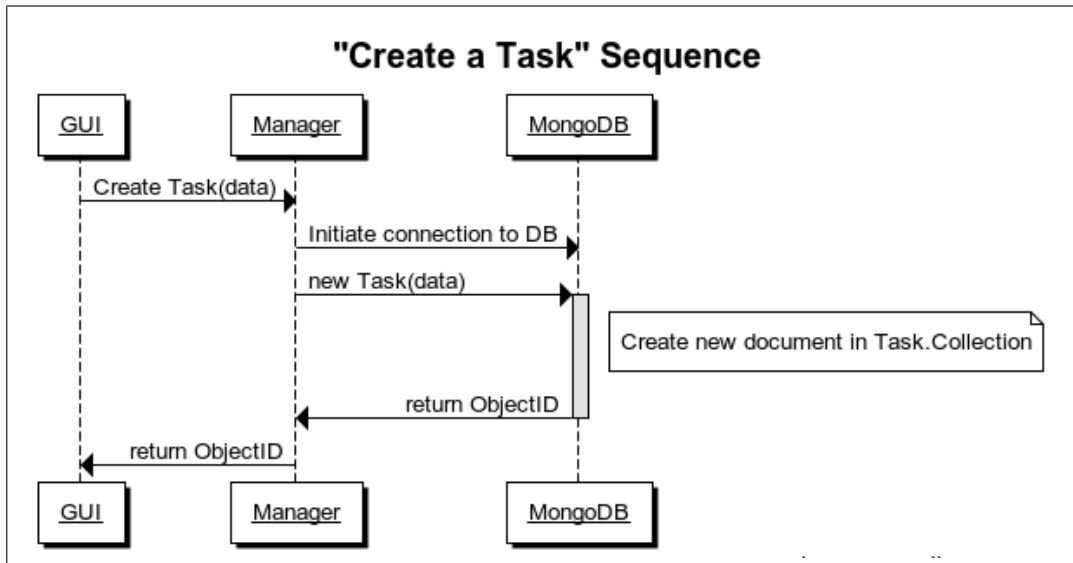


Figure 3.2: Sequence Diagram for Creating a Task

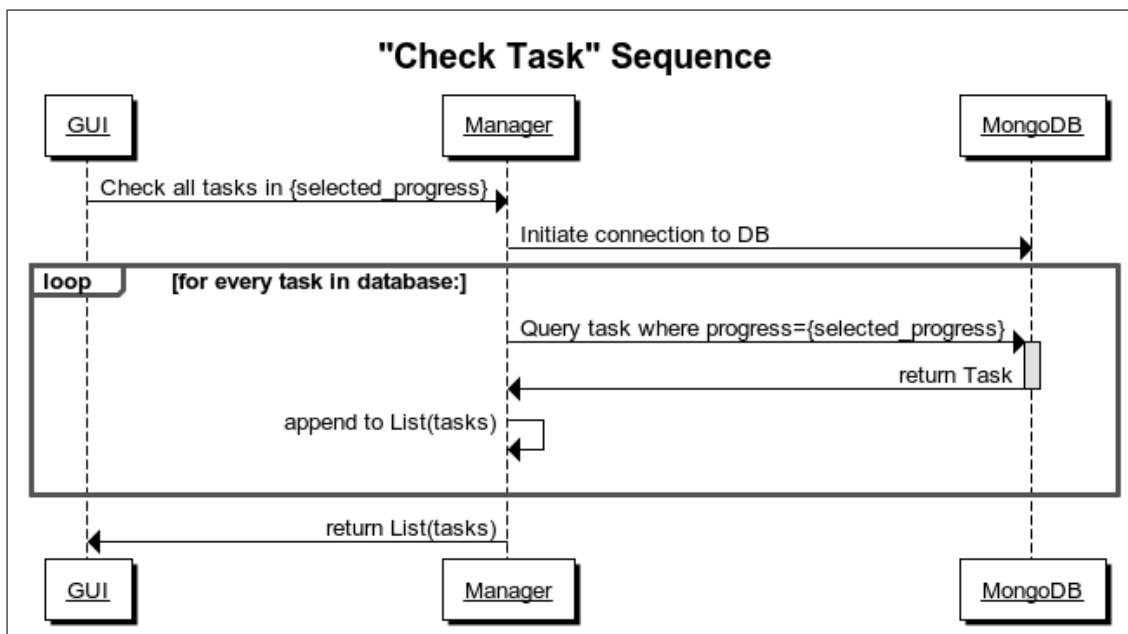


Figure 3.3: Sequence Diagram for Checking Tasks

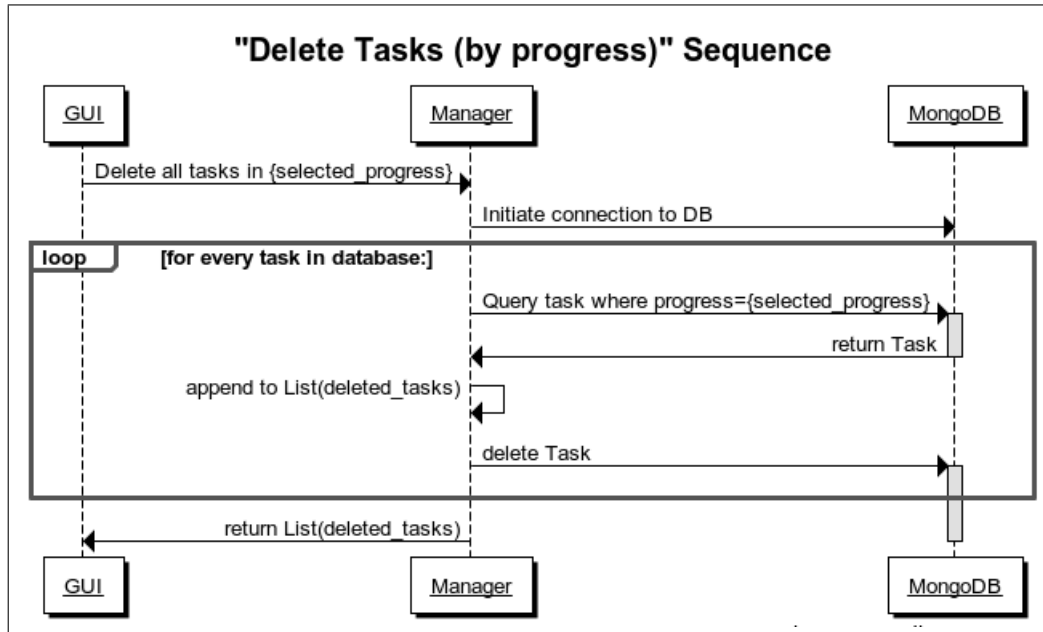


Figure 3.4: Sequence Diagram for Creating a Task by Progress

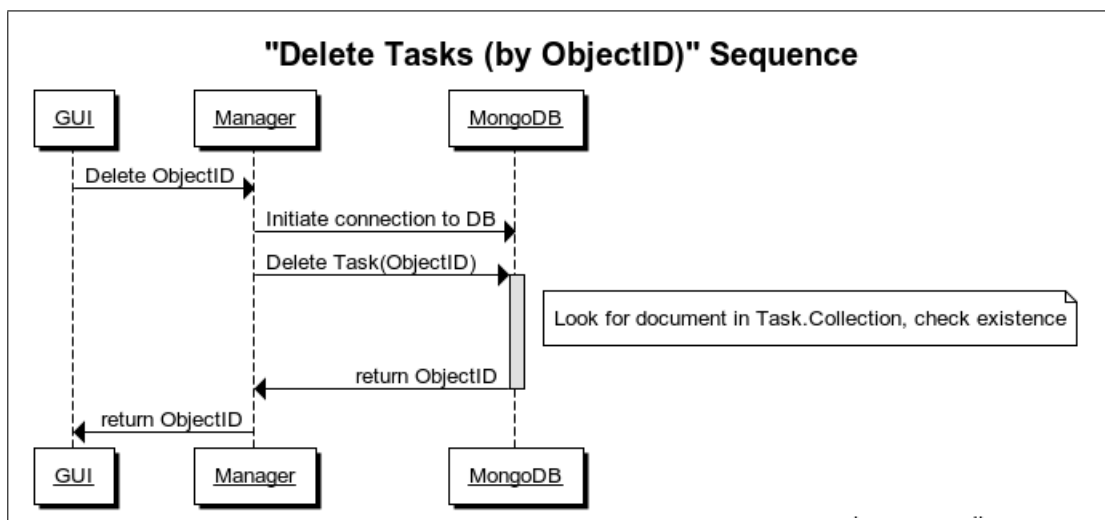


Figure 3.5: Sequence Diagram for Creating a Task by ObjectID

### 3.5.2 Cuckoo Manager

The first part of the design requires creating a web-based manager which handles task creation, checking and deletion within the database. A step-by-step outline for these functions are outlined below.

1. Start the web-server and access the GUI on localhost:8080 through a web browser.
2. To create a task, the user clicks on the *Create a Task* tab and enters all URLs for that task, specify the task's timeout as well as its priority. Once sent, the manager calls on

a script that inserts that task as a document under the `Collection.Tasks` collection in MongoDB.

3. In order to check tasks that are in a current progress state, the user can click on the *Check Tasks* tab and select one of the three states listed - idle, inprogress, completed. Once the query is sent, the manager calls a script that checks the `Collection.Tasks` collection for all tasks in that selected progress state.
4. If a user wants to delete a task, they can select the *Delete Tasks* tab and are given two options - delete a task from a given task ID or delete all tasks in either idle or completed states.
  - If a user chooses to delete a task from a given task ID, then they must provide the manager a valid, existing ID. The manager then queries the database for the ID and if it exists, removes that particular task from the `Collection.Tasks` collection.
  - If the user wants to delete tasks that are in a particular progress state, they can choose the state (idle or completed), and the manager then queries for all tasks in that selected state and removes each task from the `Collection.Tasks` collection.

Figure 3.3 illustrates the main layout of the manager's GUI.

The figure displays three wireframes of the 'Zombie Beatdown Manager' GUI, each with a header bar containing the title and three navigation links: 'Create a Task', 'Check Tasks', and 'Delete Tasks'.

**Wireframe 1: Create a Task**  
This wireframe shows a form titled 'Create a Task' with a 'Form' label. It contains three input fields:

- URL: list of urls
- Timeout: numerical value
- Priority: Low, Normal, High

**Wireframe 2: Check Tasks**  
This wireframe shows a form titled 'Check Tasks' with a 'Form' label. It contains one input field:

- Task Progress: idle, inprogress, completed

**Wireframe 3: Delete Tasks**  
This wireframe shows a form titled 'Delete Tasks' with a 'Form' label. It contains two input fields:

- Task Progress: idle, completed
- ObjectID: ObjectID

Figure 3.6: Wireframes of the three web pages for the Zombie Beatdown Manager



The manager's GUI contains a form on each page and each form is unique for the purpose of that particular function. For example, when creating a task, every data entered in the form is extracted by the manager, using unique identifiers for each field, and passed on to the database.

### 3.5.3 System Analysis

The second part of the system is the existing Cuckoo system and is responsible for pulling tasks from the database and managing available virtual machines to analyse the URLs in a task. In order to satisfy the first design requirement outlined in section 3.2, it was important to ensure that the existing design and functionality of Cuckoo is not altered and that a user can alternatively use the sandbox in its original state if they wanted to. To do so, the system was extended, rather than modified, to call on Cuckoo's main script, `cuckoo.py`, for each individual URL in a task. The extension, `worker.py`, has the following main functions - pull the most prioritized task from the database, parse through the URLs and for each one, run the `cuckoo.py` script to analyse the URL using an available minion. A high-level description of the worker's role is outlined below and a sequence diagram is presented in Figure 3.7.

1. Pull the highest prioritized task from the database by running the `worker.py` script from the worker, through the command line.
2. For each URL, the script will then add each one to the queue for analysis and saves all the task IDs (of URLs) in a list which is also sent to the database.
3. Once the analysis begins, the worker checks whether the task is completed every five seconds. For each iteration, it observes whether the size of the `tid_list` (obtained from Step 2), is the same as the number of tasks that are completed.
4. If all URLs are finished being analysed for a task, the user is notified of its process and the status is updated in the database from "inprogress" to "completed".

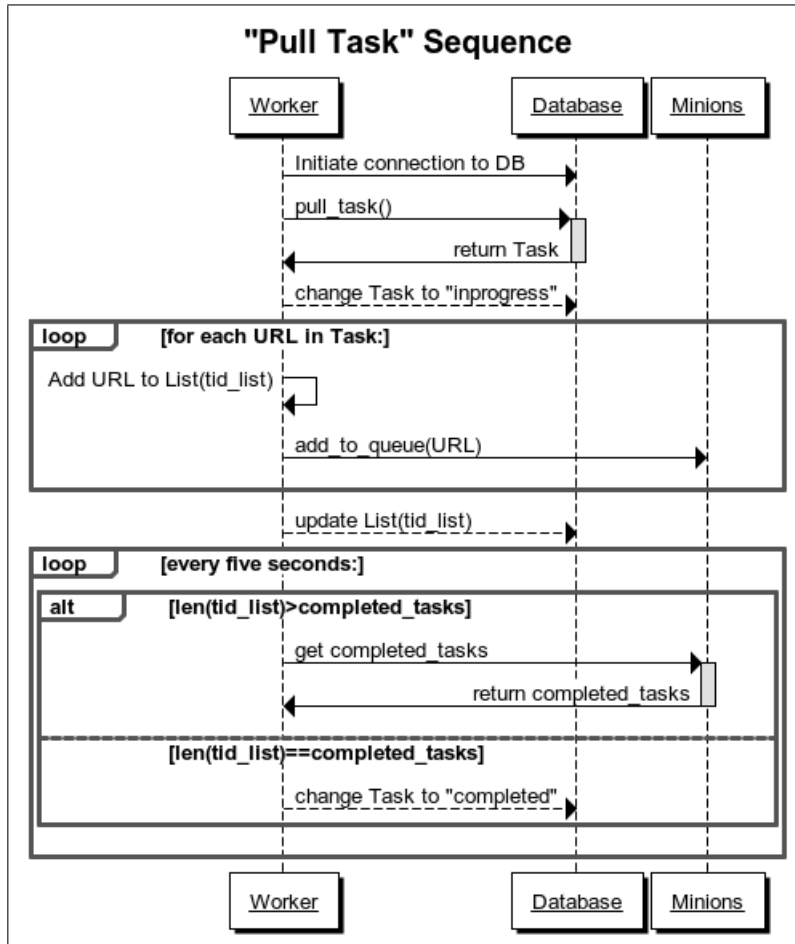


Figure 3.7: Sequence Diagram for Pulling a Task

### 3.5.4 Satisfying Design Requirements

The design requirements discussed in Section 3.2 acted as guidelines for the system's design and we were able to approach the problem with a solution that satisfies that criteria. First, by extending the functionality of Cuckoo rather than modifying it, we were able to maintain the existing functionalities of the sandbox and won't inconvenience users that want to use it in its original state. This means that we won't lose regular users who want to analyse individual URLs but make it relatively easy for other users who would have a bigger workload of analysing multiple URLs. Furthermore, the ability to allow workers, who are geographically distributed, to pull tasks, based on priority, from the database and analyse the URLs included in that task also satisfies other design requirements for the system.

In the manager-side, the fifth and sixth requirements were satisfied as the system was designed to achieve a fully decoupled distributed system and enables the master to create, check and delete tasks from the database, even without any active workers available. By doing so, the manager doesn't become responsible for maintaining and monitoring workers but is only concerned about pushing tasks to the database and making sure that there is work available for workers. This also means that minions can come and go within the system and changes are not required in the manager-side as it doesn't get affected. This is another requirement outlined in Section 3.2 that was satisfied by our system design.

## Chapter 4

# Implementation

This chapter gives technical details about the implementation of each major component discussed in the design chapter. First the environment setup and tools used for the implementation are outlined. Details about the Zombie Beatdown's scripts are then discussed, providing insight on task management around the database followed by a discussion on Zombie Beatdown's workflow process, giving a step-by-step outline of how it creating a task, pulling it from the database before finally analyzing the URLs.

### 4.1 Development Environment

During the implementation of Zombie Beatdown, there are four primary tools that were used for the development, Eclipse IDE, VirtualBox, MongoHQ and Github. Furthermore, the extension was written in Python and a justification of why this was the language chosen is also discussed in this section.

#### 4.1.1 Python

Python was the programming language chosen for the one main reason - Cuckoo is all written in Python, which made it easier to extend functionalities in the same language. Furthermore, Python has an extensive collection of modules, available for import, that can be utilized to perform certain functions without having to over-complicate the scripts.

#### 4.1.2 Eclipse IDE and Python Libraries

All programming for the extension was completed using PyDev on Eclipse Python IDE's Luna Release version. Additional plugins like, EGit was also installed, although not strictly required, to help with synchronization to repositories in Github. In addition to using Eclipse Luna as an editor, other Python libraries were required to be installed in the system in order to run and test Cuckoo. These are libraries such as, Sqlalchemy, Bson, Jinja2, Magic, Pydeep, Pymongo, Yara, Libvirt, Bottlepy, Django, Pefile, Volatility, MAEC and Chardet.

### 4.1.3 VirtualBox

VirtualBox was used as the virtualization software package to load and run the Windows virtual machines, performing URL analysis. Although Cuckoo supports VirtualBox, KVM and VMWare, I decided to use VirtualBox because of its open-source nature and installability.

### 4.1.4 MongoHQ

Since it was decided to use MongoDB as the database to store tasks, MongoHQ was used as the platform to deploy and host the database. With MongoHQ's free "Sandbox" account, 512MB of data is able to be stored, allows multiple user access as well as easy database management. This was enough to host the tasks for my system since each task, with around 10 URLs each, is approximately 5KB which means a 512MB database would be able to store over 100,000 tasks.

### 4.1.5 GitHub

Github was the Git repository web-based hosting service used for version control and code management since it allowed a secure and easy platform for me to store and access my code from anywhere. Having all the code saved in a hard drive or a laptop risks data loss or corruption, thus, GitHub was utilized to manage the code for Zombie Beatdown.

## 4.2 Zombie Beatdown Manager

The main challenge with the existing implementation of Cuckoo is its inability to process more than one URL at a time, which means any user wanting to process a set of URLs and analyze them would be required to add each URL to the queue individually. This can be an inconvenience, especially for those processing more than 10 URLs at a time. The main purpose of extending Cuckoo with an added functionality of being able to automate the processing of multiple URLs is to make it significantly easier for a user to overcome this exact challenge. This new functionality will be made of two main components - the existing Cuckoo sandbox and a manager.

As discussed earlier in Chapter 3, the manager has three main functions - create a task, check its status or delete tasks - and is the main interface that a user interacts with in order to push tasks to the database. This is achieved by running an HTTP web server using Python, HTML and a RESTful API, and creating handlers for GET and POST requests to send data to the database by passing parameters. Figure 4.1 presents a table summarizing the method, URL and result for each GET and POST requests.

Method	URL	Parameters	Result
POST	/create	list(urls), priority, timeout	Creates a new task and saves to database
	/check	priority	Pulls all tasks from the database that matches the given progress
	/delete	ObjectId	Deletes the passed ObjectId from the database
	/delete	progress	Deletes all tasks with that match the passed progress from the database
GET	/create	None	Checks that create.html exists and loads it
	/check	None	Checks that check.html exists and loads it
	/delete	None	Checks that delete.html exists and loads it

Figure 4.1: RESTful Service API for Manager's web server

### 4.2.1 Scripts

There are two main scripts that make up the extension for the manager-side of Zombie Beatdown - `servertest.py` and `mongo.py`.

#### Servertest.py

`Servertest.py` is a Python script that contains the handlers for POST requests and is the main script that a user runs to gain access to the manager under `localhost:8080`. Depending on which page the user clicks on - *create*, *check* or *delete* - the handler will load the corresponding HTML file and display the page on the browser. Once a page is displayed, the user may fill in the form for that page and click on *Send*. The HTML page will then call on the POST request for that particular page which sequentially prompts the handler to execute the commands for that corresponding page. For example, when clicking *Send* on the "Create a Task" page, a POST request for `/send` will be sent to the server which will prompt `servertest.py` to execute a set of commands as shown on Figure 4.2.

```

43     # Handler for the POST requests
44     def do_POST(self):
45         if self.path == "/send":
46             form = cgi.FieldStorage(
47                 fp=self.rfile,
48                 headers=self.headers,
49                 environ={'REQUEST_METHOD': 'POST',
50                        'CONTENT_TYPE': self.headers['Content-Type'],
51                 })
52
53             data = form.getvalue("urls")
54             urls = [x.strip() for x in re.split(' |,|\n', data)]
55             timeout = form.getvalue("timeout")
56             priority = form.getvalue("priority")
57
58             t = Task(urls, timeout, priority)
59             self.send_response(200)
60             self.end_headers()
61
62             obj_id = mongo.push_task(t)
63             self.wfile.write("Task pushed to DB successfully.\nObjectID is %s" % obj_id)
64             return

```

Figure 4.2: Handler for POST request `/send`

From Figure 4.2, we see that once a POST request is sent, all information that the user entered and included in the form is extracted and saved in a field. The URLs are then parsed and stripped individually to a list and is used to create a new Task object along with the extracted timeout and priority from the form. Once a Task object is created, `do_POST()` will then push the object to the database by calling another function in the `mongo.py` script.

Both functions, *Check* and *Delete* also work in a similar way - once their corresponding HTML file is prompted to call on the handler for `/check` and `/delete`, a set of commands are executed. When checking tasks based on their status, `servertest.py` first ensures that a progress was selected, then calls on a function in `mongo.py` that returns and prints all tasks with that selected progress. In a similar way, deleting tasks by status works exactly the same but rather than returning a list of tasks, it only returns the list of the ObjectIDs for tasks that have been deleted. However, if a user were to delete a task by ObjectID, the ID is extracted from the form and send to a function in `mongo.py` which deletes that one particular task. `Servertest.py` basically works as a translator between the user and the scripts which manage the tasks in the database.

## Mongo.py

`Mongo.py` handles all functions that deal with the database directly. It is the only script in the manager-side that actually communicates and makes changes to the database. All four main functions - creating a task, checking task status, deleting tasks by progress or deleting task by ID - are all defined in this script and each create changes in the database's `Collection.tasks` collection. Before making changes to the database, the manager must first connect to the database and is defined under the `MONGO_URL` global field. Although this can be changed so that the user is prompted for database details through a login mechanism (as discussed in Conclusion 6.1.5), it was easier to hard code the database details in just the same as how `Cuckoo.conf` files are also configured manually at the beginning. Furthermore, where `servertest.py` is a translator for the manager, `mongo.py` is the messenger between the user and the database. The two interact and coordinate with each other to obtain the ability to manage and view existing tasks in the database.

```
24 def delete_progress(progress):
25     client = MongoClient(MONGO_URL)
26     db = client.ZombieBeatdown
27     collection = db.tasks
28     list = []
29     for task in collection.find({ "progress":progress }):
30         list.append(task['_id'])
31         collection.remove(task)
32     return list
```

Figure 4.3: "Delete by Progress" script in `mongo.py`

Figure 4.3 presents an example of the *Delete by Progress* function in `mongo.py`, outlining how a set of tasks with a selected progress state are deleted from the database. From here, we see how a progress is passed as a parameter, which is then used to look for all tasks that are in that progress state, before removing each one from the `Collection.tasks` collection.

## 4.2.2 GUI

The final Zombie Beatdown Manager user interface is presented in Figure 4.4. The GUI implementation was implemented purely on Eclipse Luna, using their Web Tools Platform (WTP) plugin. The manager consists of three pages, as previously outlined earlier - *create*, *check* or *delete*.

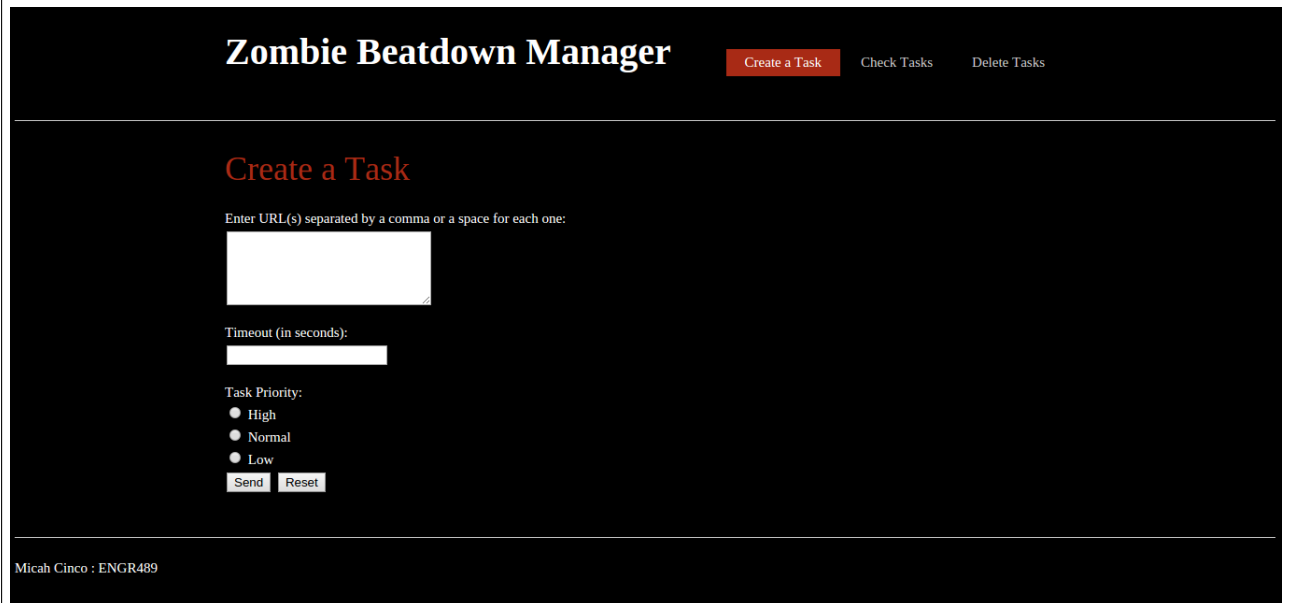


Figure 4.4: "Create a Task" web page

For all three pages, the navigation pane and footer are all consistent. The main difference is the form where the user selects or enters information in order to perform certain functions within the database. The forms consists of text fields, radio buttons and standard buttons that is each equivalent to a "value". This value is the selected radio or standard button or the information inside a text field and is the part of the form that gets extracted by `servertest.py`, before passing on to `mongo.py`. The "name" of a button, radio button or text field, is the identifier for that object. For example, if the script extracts the "value" of radio button, "priority", the value of the selected radio button is returned (see Figure 4.5).

```
37 <form method="POST" action="/check">
38   <p>
39     Task Progress: <br>
40     <input type="radio" name="progress" value="idle"> Idle<br>
41     <input type="radio" name="progress" value="inprogress"> In Progress<br>
42     <input type="radio" name="progress" value="completed"> Completed<br><br>
43
44     <input type="submit" value="Send"> <input type="reset">
45
46   </p>
47 </form>
```

Figure 4.5: HTML form code for "Check a Task" web page

## 4.3 Zombie Beatdown Worker Analysis

As discussed earlier in Chapter 3, the second part of Zombie Beatdown's extension to Cuckoo is the Worker Analysis side. This part of the system manages the pulled task and coordinates all available minions together to execute the analysis for each URL. Once all URLs in a task has been successfully analyzed (regardless of its result), the task is updated as "completed" in the database. The way that the current Cuckoo system works requires two terminal windows - one for `cuckoo.py` and another to submit URLs to the local queue using `submit.py`.

### 4.3.1 Scripts

Rather than modifying existing scripts, new ones were implemented to extend Cuckoo's existing functionalities. This extension is comprised of three main scripts - `worker.py`, `taskhandler.py` and `mongo.py`. Similar to the manager-side, the worker-side also has a `mongo.py` script which is in charge of all commands that make direct changes to the database. It's not the main script but is the main messenger between the worker and the database. Furthermore, these new scripts also make use of some of the same functions that the existing system utilizes - the only difference being how it's used to enable automation and processing of multiple URLs at a time.

#### Worker.py

Much like how `servertest.py` was the main script for the manager, `worker.py` is the equivalent in the worker-side and acts as the "master script" that gets run when utilizing the Zombie Beatdown extension. The very first task for this script is to pull a task from the database by calling the `pull_task()` function from `mongo.py`. Once a task is pulled, each URL is then added on to the local queue for analysis, where each is given a local task ID. These task IDs are added to a global variable, `tid_list`, which enables grouping and tracking of all URLs from that one task. As shown in Figure 3.7, the next phase would be to periodically check that the total size of completed analysis from that `tid_list` is equal to the list size. If so, then all URLs from that task are finished analyzing and updated in the database as "completed" by using the `task_done()` function from `taskhandler.py`.

#### Taskhandler.py

Since `worker.py` is like the "master script", it calls on other functions between `taskhandler.py` and `mongo.py`. `Taskhandler.py` is a script that handles any calls which directly interacts with the Cuckoo host's local queue. It has two main functions - `add_urls()` and `task_done()`. With `add_urls()`, a `Task` object is passed as a parameter and for each URL in that task, a new URL entry is made in Cuckoo's local queue (for processing) which returns a `task_id`. Each URL's `task_id` is saved in `texttttid_list` and is sent to the `Task` object's instance in the database. This process allows the system to track all `task_id`'s for every URL in a particular task and makes it easier to group them together in order to check whether the task status is completed (as discussed earlier).



`task_done()` is another function that is utilized to check whether a whole Task object's analysis is finished by checking every single process' state for each URL corresponding to the previously mentioned, `tid_list`. Within a Cuckoo system, an analysis' state can be one of the following: `TASK_PENDING`, `TASK_RUNNING`, `TASK_COMPLETED`, `TASK_REPORTED` and `TASK_RECOVERED`.

An analysis is concluded to be done if its current state is either `TASK_COMPLETED` or `TASK_REPORTED`, thus, `task_done()` calculates the total number of completed analysis by counting the total number of tasks in either state. This total count, `tasks_count` is then compared to the size of the `tid_list`. If `tasks_count` and `len(tid_list)` are not equal, then all of the URLs within a task are not yet completed and is recalculated again after five seconds, as outlined in `worker.py`.

## Mongo.py

The final script that makes the third of the main component for Zombie Beatdown's Worker extension is `mongo.py`. This script basically contains one major function - `pull_task()`. In this function, `mongo.py` pulls a task from the database that is in an "idle" state. Furthermore, it prioritizes the tasks which have the highest defined priority to satisfy the Design Requirement #4 in Section 3.2.

Within the function, the first command executed checks to see whether there are any tasks in the database that are in "idle" state. If not, then there are no available tasks to be analyzed. The next check looks at the task's priority by order - High, Normal, then Low. If there are no tasks with a "High" priority, it then checks for tasks that have a "Normal" priority, followed by "Low". The system doesn't have an error handling mechanism for this function since the manager specifies the priority of a task through the use of radio buttons, meaning that the chances that a task ends up with a priority outside the three options presented above, are very low.

`mongo.py` also contain two other minor functions - `update_tasklist()` and `task_done()`. `update_tasklist()` is the function responsible for updating the `tid_list` that the TaskHandler utilizes to group together a set of URLs that belong to the same Task object. On the other hand, `task_done()` is the last function used when an analysis is complete as it is responsible for changing a task's state from "inprogress" to "completed" by identifying a task based on a given `ObjectId`.

### 4.3.2 User Interface

To satisfy the requirements outlined in Section 3.2, our implementation of the extension should have a similar appearance to the existing Cuckoo system and provide the same user experience. To fulfill this requirement, a command line interface was chosen to be the main interface between the user and the system. Not only does this provide the exact user experience compared to the existing system, an experienced user would hopefully find the system easy to navigate through and utilize, thus, not causing inconvenience, but actually making multiple URL analysis easier - which is the overall project goal.

## Chapter 5

# Evaluation

This chapter presents and discusses the results obtained from testing Zombie Beatdown. First, the evaluation environment where Zombie Beatdown was tested is outlined, followed by a functional testing of the system's complete process. The functional testing is somewhat similar to a system test where the whole system's workflow is tested to verify that Cuckoo is able to successfully analyze tasks pulled from the database and that the manager is able to perform the three main functions of creating, checking and deleting tasks. The system's load balancing mechanism for pulling higher prioritized tasks is also tested as well as error handling mechanisms in the manager-side. Finally, the performance of the whole system is evaluated after the extension was integrated to check the latency when pulling tasks from that database and the job time comparisons between individually analyzing web servers and analyzing tasks of 10, 20 and 30 URLs. By doing so, we were able to quantify the existing system's performance and compare it to our implementation, highlighting improvements and discussing existing limitations that might affect the generalization of the results.

### 5.1 Evaluation Environment

The evaluation for Zombie Beatdown was done on two identical machines - one machine running the manager and another, Linux-based laptop, running the worker and minions. The machines use an Intel Core i7-2640M CPU processor running at 2.80GHz and 8GB of RAM. The laptop running the manager is Windows-based and ran the manager locally, managing tasks in the database and is connected to the same local area network as the worker.

### 5.2 Functional Testing

This section tests the functionality of Zombie Beatdown's manager to ensure that the requirements outlined in this document is and that each step outlined in the flow of execution of this document functions as described. To complete this white-box testing stage of the project, I collected a list of valid web servers from [?] for the minions to analyze when a worker pulls a task. This will allow us to observe the system functionality in a non-simulated environment.

### 5.2.1 Zombie Beatdown Manager - Task Management

In order to test that the manager functions as expected, a set of manual tests were executed to observe the system's behavior.

#### Test Case 1: Task Creation

To test that the manager can create a task using the GUI, two tasks were created - one with 20 URLs and another with 10 URLs. The sets of URLs will then be passed to the database with or without selected "priority" and "timeout" values. Without any "priority" and "timeout" values entered in the GUI, the system was still able to create a task and send it to the database using default values - priority as "Normal" and timeout at "100". This allows the system to avoid creating tasks that are not valid or does not have enough required information to be pulled by a worker and causing errors. Furthermore, there were no behavioral differences observed between pushing 20 and 10 tasks to the database. When a task has been successfully created, a message is returned to the user as shown in Figure 5.1 which gives the task's *ObjectId*.

```
Task pushed to DB successfully.  
ObjectID is 543ecc3099460b20641e940a
```

Figure 5.1: Message returned after successful task creation

Figure 5.2 shows that after the task was created, the database reflects the newly created task with all the data passed by the manager.

```
{  
  _id: ObjectId("543ecc3099460b20641e940a"),  
  urls: [  
    "http://www.imgur.com/",  
    "http://www.163.com/",  
    "http://www.google.it/",  
    "http://www.imdb.com/",  
    "http://www.google.es/",  
    "http://www.t.co/",  
    "http://www.adcash.com/",  
    "http://www.aliexpress.com/",  
    "http://www.amazon.co.jp/",  
    "http://www.go.com/",  
    "http://www.alibaba.com/",  
    "http://www.craigslist.org/",  
    "http://www.xhamster.com/",  
    "http://www.google.com.mx/",  
    "http://www.stackoverflow.com/",  
    "http://www.fc2.com/",  
    "http://www.google.ca/",  
    "http://www.bbc.co.uk/",  
    "http://www.espn.go.com/",  
    "http://www.cnn.com/"  
  ],  
  priority: "Normal",  
  timeout: "120",  
  date_created: "2014-10-16 08:34:07.744537",  
  progress: "idle",  
  ..progress: r..  
}
```

Figure 5.2: Database after successful task creation

## Test Case 2: Task Checking

To test that the manager is able check all tasks in a particular state using the GUI, we ensured that there were tasks in the database at different states and depending on the progress that the user selected, should return the set of tasks that are in that particular state. For this test, there were 59 tasks in the database, where two tasks are in "idle" state, one in "inprogress" and 56 in "completed" state. For each progress state, the manager was able to return the correct tasks and this was validated by querying the database from MongoHQ's web interface to the tasks in that state. Figure 5.3 shows what was returned to the manager when the user selected to check all tasks that are in "idle" state.

```
[
  {
    "_id": {
      "$oid": "543f907399460b2c48ccfac6"
    },
    "date_created": "2014-10-16 22:31:30.183408",
    "priority": "Low",
    "progress": "idle",
    "tid_list": [],
    "timeout": "120",
    "urls": [
      "www.facebook.com",
      "www.gmail.com",
      "www.trello.com",
      "blackboard.ecs.vuw.ac.nz",
      "ecs.vuw.ac.nz",
      "google.com",
      "westpac.co.nz"
    ]
  },
  {
    "_id": {
      "$oid": "543f90c899460b2c48ccfac7"
    },
    "date_created": "2014-10-16 22:32:55.674752",
    "priority": "Normal",
    "progress": "idle",
    "tid_list": [],
    "timeout": "300",
    "urls": [
      "www.facebook.com",
      "www.gmail.com",
      "www.trello.com",
      "blackboard.ecs.vuw.ac.nz",
      "ecs.vuw.ac.nz",
      "google.com",
      "westpac.co.nz"
    ]
  }
]
There are 59 tasks in the database and 2 are in "idle" status.
```

Figure 5.3: Tasks returned when checking for "idle" tasks

## Test Case 3: Task Deletion by ObjectId

Deleting a task from the database through the manager's GUI is done in two different ways, deleting by the task's ObjectId and deleting all tasks in a particular progress state. To test that the manager can delete a task using its ObjectId, I passed the ObjectId of one task in the database and sent the request. This reported back that the task with that given ObjectId was successfully deleted.

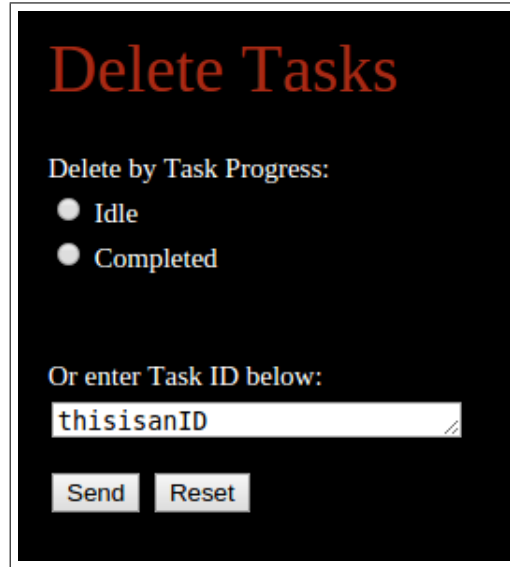


Figure 5.4: Manager GUI when deleting task by ObjectId

However, if a user enters an invalid ObjectId as shown in Figure 5.4, then the manager would report back to the user that the ObjectId entered is not valid. Furthermore, if the ObjectId entered is valid but not in the database, the manager would report back that it cannot find that particular ObjectId.

#### Test Case 4: Task Deletion by Progress

To test that the manager can delete tasks in a particular progress state, one of the two states - "idle" or "completed" - was selected from the GUI and passed to the manager to send the request to the database. This reported back that all tasks in the database in "completed" state was deleted successfully. Furthermore, if a user does not enter an ObjectId or progress state and clicked on "Send", the GUI would report back that there was no progress or ObjectId entered. This error handling mechanism ensures that the correct values are passes when sending the query to the database to avoid unexpected errors.

Table 5.1 below gives a summary of the three test cases described above along with the results that each test case acquired.

Test Case	Result
1	Pass
2	Pass
3	Pass

Table 5.1: Results for Task Management Test Cases

#### 5.2.2 Zombie Beatdown Worker - Task Analysis

To test the Worker-side of Zombie Beatdown, the test cases were split in two halves - one to test the workload balancing mechanism when pulling tasks, and another to test that the

task is being allocated to an available minion and analyzed correctly.

## Test Case 1: Pulling Tasks from the database

To test that the Worker is able to pull tasks from the database using the load balancing scheme in place, a set of tasks were added to the database that had different priorities. Figure 5.5 shows four tasks in the database, where one task (on top) is currently “inprogress” with a priority of “High”. Looking at the other tasks, the second task also with “High” priority should be the next task to be pulled once the first one finishes, not one of the two tasks below with a “Normal” priority.

<pre>{_id: ObjectId("543ce89b99460b594209b9ae"), date_created: "2014-10-14 22:10:50.169421", priority: "High", progress: "inprogress", tid_list: [ 93, 94, 95, 96, 97, 98, 99, 100, 101, 102 ], timeout: "180", urls: [ "http://www.media.tumblr.com/", "http://www.hdfcbank.com/", "http://www.tubecup.com/", "http://www.ebay.in/", "http://www.ad120m.com/", "http://www.rediff.com/", "http://www.douban.com/", "http://www.ifeng.com/", "http://www.gogorithm.com/", "http://www.github.io" ] }</pre>
<pre>{_id: ObjectId("543ce89e99460b594209b9af"), date_created: "2014-10-14 22:10:53.544274", priority: "High", progress: "idle", tid_list: [], timeout: "180", urls: [ "http://www.media.tumblr.com/", "http://www.hdfcbank.com/", "http://www.tubecup.com/", "http://www.ebay.in/", "http://www.ad120m.com/", "http://www.rediff.com/", "http://www.douban.com/", "http://www.ifeng.com/", "http://www.gogorithm.com/", "http://www.github.io" ] }</pre>
<pre>{_id: ObjectId("543ecc3099460b20641e940a"), urls: [ "http://www.imgur.com/", "http://www.163.com/", "http://www.google.it/", "http://www.imdb.com/", "http://www.google.es/", "http://www.t.co/", "http://www.adcash.com/", "http://www.aliexpress.com/", "http://www.amazon.co.jp/", "http://www.go.com/", "http://www.alibaba.com/", "http://www.craigslist.org/", "http://www.xhamster.com/", "http://www.google.com.mx/", "http://www.stackoverflow.com/", "http://www.fc2.com/", "http://www.google.ca/", "http://www.bbc.co.uk/", "http://www.espn.go.com/", "http://www.cnn.com/" ], priority: "Normal", timeout: "120", date_created: "2014-10-16 08:34:07.744537", progress: "idle", tid_list: [] }</pre>
<pre>{_id: ObjectId("543ecc7499460b20641e940b"), urls: [ "http://www.imgur.com/", "http://www.163.com/", "http://www.google.it/", "http://www.imdb.com/", "http://www.google.es/", "http://www.t.co/", "http://www.adcash.com/", "http://www.aliexpress.com/", "http://www.amazon.co.jp/", "http://www.go.com/", "http://www.alibaba.com/", "http://www.craigslist.org/", "http://www.xhamster.com/", "http://www.google.com.mx/", "http://www.stackoverflow.com/", "http://www.fc2.com/", "http://www.google.ca/", "http://www.bbc.co.uk/", "http://www.espn.go.com/", "http://www.cnn.com/" ], priority: "Normal", timeout: "120", date_created: "2014-10-16 08:35:09.969054", progress: "idle", tid_list: [] }</pre>

Figure 5.5: Pulling of Tasks with “High” priority

Figure 5.6 shows the database after the next task was pulled and we see that the workload balancing mechanism works correctly since the second task, with a “High” priority, was pulled and is currently “inprogress”.

<pre>{_id: ObjectId("543ce89b99460b594209b9ae"), date_created: "2014-10-14 22:10:50.169421", priority: "High", progress: "completed", tid_list: [ 93, 94, 95, 96, 97, 98, 99, 100, 101, 102 ], timeout: "180", urls: [ "http://www.media.tumblr.com/", "http://www.hdfcbank.com/", "http://www.tubecup.com/", "http://www.ebay.in/", "http://www.ad120m.com/", "http://www.rediff.com/", "http://www.douban.com/", "http://www.ifeng.com/", "http://www.gogorithm.com/", "http://www.github.io" ] }</pre>
<pre>{_id: ObjectId("543ce89e99460b594209b9af"), date_created: "2014-10-14 22:10:53.544274", priority: "High", progress: "inprogress", tid_list: [ 103, 104, 105, 106, 107, 108, 109, 110, 111, 112 ], timeout: "180", urls: [ "http://www.media.tumblr.com/", "http://www.hdfcbank.com/", "http://www.tubecup.com/", "http://www.ebay.in/", "http://www.ad120m.com/", "http://www.rediff.com/", "http://www.douban.com/", "http://www.ifeng.com/", "http://www.gogorithm.com/", "http://www.github.io" ] }</pre>
<pre>{_id: ObjectId("543ecc3099460b20641e940a"), urls: [ "http://www.imgur.com/", "http://www.163.com/", "http://www.google.it/", "http://www.imdb.com/", "http://www.google.es/", "http://www.t.co/", "http://www.adcash.com/", "http://www.aliexpress.com/", "http://www.amazon.co.jp/", "http://www.go.com/", "http://www.alibaba.com/", "http://www.craigslist.org/", "http://www.xhamster.com/", "http://www.google.com.mx/", "http://www.stackoverflow.com/", "http://www.fc2.com/", "http://www.google.ca/", "http://www.bbc.co.uk/", "http://www.espn.go.com/", "http://www.cnn.com/" ], priority: "Normal", timeout: "120", date_created: "2014-10-16 08:34:07.744537", progress: "idle", tid_list: [] }</pre>
<pre>{_id: ObjectId("543ecc7499460b20641e940b"), urls: [ "http://www.imgur.com/", "http://www.163.com/", "http://www.google.it/", "http://www.imdb.com/", "http://www.google.es/", "http://www.t.co/", "http://www.adcash.com/", "http://www.aliexpress.com/", "http://www.amazon.co.jp/", "http://www.go.com/", "http://www.alibaba.com/", "http://www.craigslist.org/", "http://www.xhamster.com/", "http://www.google.com.mx/", "http://www.stackoverflow.com/", "http://www.fc2.com/", "http://www.google.ca/", "http://www.bbc.co.uk/", "http://www.espn.go.com/", "http://www.cnn.com/" ], priority: "Normal", timeout: "120", date_created: "2014-10-16 08:35:09.969054", progress: "idle", tid_list: [] }</pre>

Figure 5.6: 2nd Pull for Task with “High” priority

In addition to the workload balancing mechanism, if all tasks in the database are completed or none are in an “idle” state, then the database returned “None” and a message was printed to let the Worker know that this was the case.



## 5.3 Performance Testing

This section will test the performance impact of the Zombie Beatdown extension to Cuckoo's average job time when analyzing multiple web servers, rather than doing so individually, as it is in its current implementation, as well as the latency for pulling a task from the database.

Performance metrics such as latency and average job time are important to evaluation since in a real world environment, URLs that are being analyzed can be up to 500,000 web servers to visit. If Zombie Beatdown's extension to Cuckoo does not improve the performance of the system, which was what we have set out to do for this project, users may be deterred from using the added functionality.

### 5.3.1 Network Latency

Due to the fact that multiple workers are expected to be pulling tasks from the database simultaneously, we wanted to see the latency for performing this task. Since a task is expected to contain, typically, 20 to 30 URLs, to test the latency induced from fetching a task, we conducted a test which collects the round trip delay time from creating a query to the database, until the worker receives a task in return from the database.

Round trip latency was observed from the worker-side and was done by using one of Python's `time` library which allowed us to measure the time between the initiation of a query until when the expected document is received. This approach was taken due to the fact there were no functions, provided by MongoHQ or `pymongo`, which returned round trip delay time or network latency. For this test, 30 samples were collected from pulling a tasks of 20 and 30 URLs each. Figure 5.9 presents a plot of the statistics gathered from latency induced through pulling a task on the worker-side for varying task sizes.

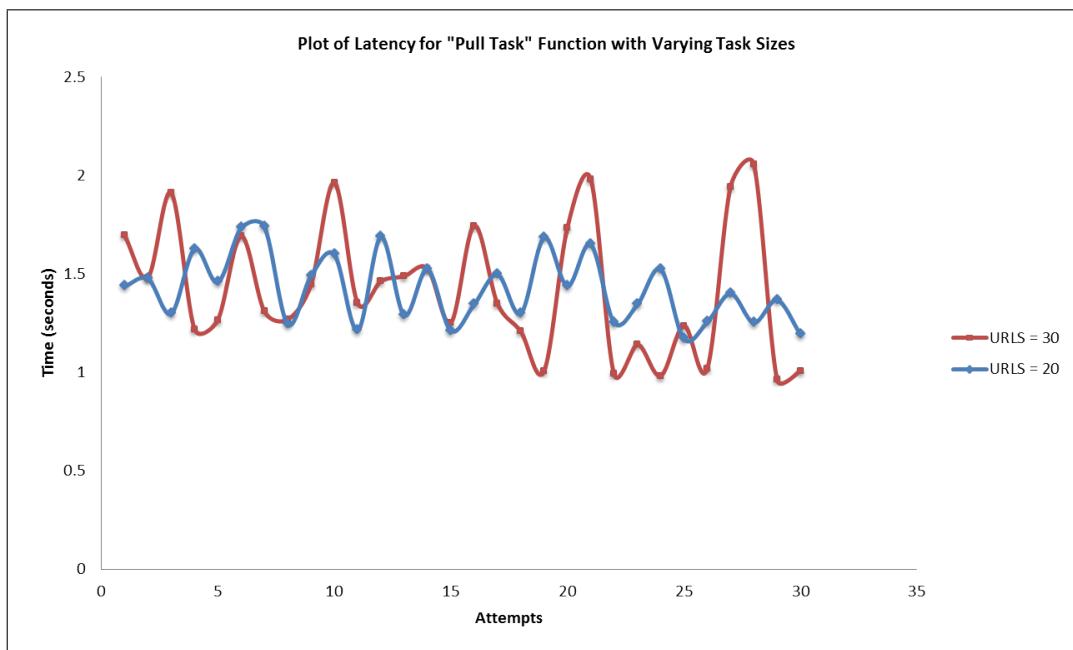


Figure 5.9: Plot of Latency for "Pull Task" function with Varying URL Size



Here, we see that to pull a task, which is approximately 5KB each, the network takes 1.42 seconds, on average, for both 20 and 30 tasks. However, it's hard to say whether the sample collected is a true representation of the population. According to the Central Limit Theorem, the distribution of all sample means is approximately *normal*, regardless of the underlying distribution. Thus, we can deduce that the sample mean collected is approximately equal to the unknown population mean. Considering that our data is made of 30 samples and is considered to be normally distributed (based on the Central Limit Theorem), I decided to measure the Z-distribution, which is a probability dense function, of the data to measure the level of confidence in our estimations.

To obtain a 95% confidence interval for our estimations, meaning that we are 95% sure that our unknown population mean is between a calculated lower and upper limit, the standard deviation of the results were obtained. The formula shown below was then utilized to calculate the lower and upper bound for each set of results, where  $\bar{x}$  is the sample mean,  $\sigma$  is the standard deviation of the sample and  $n$  is the sample size.

$$\bar{x} \pm 1.96 \sigma / \sqrt{n}$$

Table 5.3 presents the latency comparisons for the results collected for pulling 20 and 30 URLs in a task. Based on the results, we see that the latency for pulling a task with 20 or 30 web servers are almost identical and have no significant difference in time.

Task Size	Results
20 URLs	1.4235 +/- 0.0623 secs
30 URLs	1.4240 +/- 0.1204 secs

Table 5.3: 95% Confidence Summary on Collected Latency Data

### 5.3.2 Average Job Time

To address the design requirements for this project, the Zombie Beatdown extension must not only allow a user to analyze multiple web servers in one script execution, it must also be able to do so in a reasonable amount of time. Much like testing the application latency, testing for the average job time was done by pulling a task from the database and analyzing the URLs within that task by allowing the script to run and allocate tasks to available minions. For these set of tests, 10, 20 and 30 URLs were tested within a tasks, 30 times to ensure that the sample is normally distributed. Furthermore, since there was no way of comparing these new metrics with the existing system's performance, I wrote a script that would run and analyze 10 URLs individually, and measure the job time to be able to quantify the existing system's performance. The sample of 30, allows us to compare the sample means with each other to gain knowledge on how Zombie Beatdown's extension was able to impact the performance of task analysis.

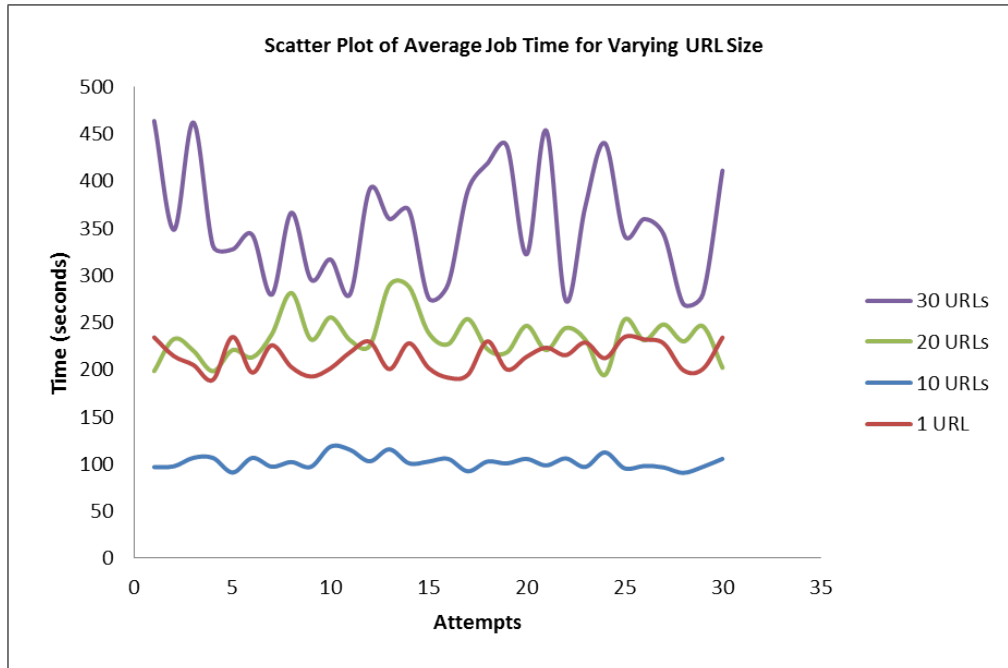


Figure 5.10: Plot of Average Job Time for Varying URL Size

Figure 5.10 plots the average job time for each task with varying URL sizes. It also presents the job time for analyzing 10 URLs individually. From the graph, we see that as the number of URLs increase, using the Zombie Beatdown extension allows a more efficient way of processing multiple URLs at a time rather than individually which takes approximately 213 seconds, or almost four minutes, to process 10 URLs. On the other hand, processing 10 URLs through Zombie Beatdown is able to be completed in approximately 102 seconds or roughly a minute and a half.

The variance also difference dramatically as the number of URLs increase in size. When only processing 10 URLs per task, the results were all very similar whereas once the URLs started increasing, results were fluctuating up and down constantly. Much like the application latency, there are no known population means for the results, thus, a 95% confidence interval must be calculated in order to gain knowledge on how big the range is and what the upper and lower limits are for collected means. By calculating the z-distribution for each of the results, we are able to gain knowledge on whether we are 95% sure that our estimates are a true representation of the real population. Results of the confidence summary is outlined below on Table 5.4.

From the results, we see that as the number of URLs increase, so does the overall job time. However, looking at the overall job time for processing 10 individual URLs, one at a time, I investigated further as to why or what could cause it to take longer. From the previous tests on application latency, we know that increase in URL size does not significantly slow down the job time but through reverse engineering Cuckoo's architecture, it was found that every time a task is pushed to the local queue and allocated to an available minion, the session overhead costs and time adds adds on to the overall job time. With Zombie Beatdown, session overhead costs are decreased since tasks are pushed to the local queue once, at the beginning of task analysis, whereas by executing the URLs individually, session overhead costs add on every single time.

Task Size	Results
1 URL	213.57 +/- 5.51 secs
10 URLs	101.68 +/- 2.53 secs
20 URLs	234.13 +/- 8.59 secs
30 URLs	346.97 +/- 22.98 secs

Table 5.4: 95% Confidence Summary on Collected Job Time Data

## 5.4 Limitations of Zombie Beatdown

One major limitation for Zombie Beatdown is its dependency on the user's network and Internet connection. Since there is a lot of interaction between the worker or manager machines and the database, there is a heavy dependency on the machine's network connectivity. A slow connection would obviously affect the system's performance and may cause a user to deter from using the system.

Another limitation is the system's dependency on its hardware specifications. In the environment that I tested and executed my evaluation, there were four cores in the system and three virtual machines, which meant that each machine was able to obtain a core, leaving another for the physical Host machine, a.k.a the Worker. Obviously, a Cuckoo system with a powerful machines are able to benefit from the Zombie Beatdown extension however, a less powerful machine may be at a disadvantage and can greatly impact the system's performance.

## 5.5 Evaluation Summary

This chapter has covered two main sections of testing; functional testing and performance testing. In the functional testing, the system workflow was tested as a whole, checking to see that all main functions on the manager side are working as expected as well as all workload balancing and error handling on both Zombie Beatdown Management or Worker side.

During the performance evaluation, Zombie Beatdown was tested for application latency when pulling tasks from the database as well as the average job time for task analysis.

Overall, the evaluation has shown that Zombie Beatdown has successfully met all the design requirements outlined in Section 3.2, allowing analysis of multiple web servers without causing noticeable impact on the system's functionality.

## Chapter 6

# Conclusion and Future Work

With the rise of drive-by downloads and the dangers that it imposes on vulnerable web browsers, the improvement and enhancement of honeypots have seen a significant growth. With the limitations that Capture-HPC has in terms of compatibility with newer versions of Windows, the integration of Capture's management function with Cuckoo has been the main focus of this project. By doing so, Cuckoo obtains the capability of analyzing multiple web servers at a time, improving its current state where URLs are processed individually.

On top of successfully implementing the integration of a manager functionality to Cuckoo, this project also provided a background review which includes a literature review of existing management systems, different architectures applicable to such systems and an analysis of fault detection approaches. Constraint of the project was also investigated as well as design requirements, approaches considered for the project and the final overall architecture chosen was justified. Chapter 4 gives insight on the implementation that went into the project, detailing the different scripts and extensions made to Cuckoo. Finally, the whole system was evaluated showing functional and performance test results.

### 6.1 Project Contributions

The main contributions of this project are as follows:

- A detailed background review was produced which outlines the various architectures and designs for management systems. The review discusses the different architectures for modeling distributed management systems that currently exist in the market and it was found that Capture-HPC and a Master-Slave approach were the most relevant models applicable for comparison since Cuckoo's system is quite similar. Furthermore, a brief investigation for fault detection schemes was also completed to gain knowledge on the available models that can be used to implement the extension for Cuckoo.
- From the background review, a set of requirements were defined, extracted from the most applicable features within different architectures and systems investigated. These requirements were drawn from the observations of the strengths and weaknesses of current solutions.

- A distributed task management system named Zombie Beatdown was designed and implemented to address these requirements. Furthermore, an extension of Cuckoo was designed as part of Zombie Beatdown which allows the system to process multiple URLs at a time which allowed me to achieve what I had set out to do for this project.
- The extension and newly implemented manager, Zombie Beatdown, was implemented and evaluated. The evaluation included a testing of all the main functionalities, covering task creation, monitoring and deletion, as well as a performance testing of the system which showed improvement in speed when analyzing sets of URLs rather than individually.

## **6.2 Future Work**

### **6.2.1 Improved Fault Detection Scheme**

Currently, the Zombie Beatdown's system is dependent on Cuckoo's existing fault detection scheme. In the future, we want to be able to improve this by implementing a task handler outside the core Cuckoo architecture which would allow it to monitor tasks as well as the actual physical Host or Worker, in case that machine fails.

### **6.2.2 Result Reporting Visualization**

Cuckoo could also be extended by implementing a visualization mechanism for reports. Currently, when an analysis is finished, all files are saved in the Worker's local memory. However, if a visualization component was implemented that allows the manager to view the tasks in graphs, then the management side of Cuckoo may also be improved.

### **6.2.3 Analysis of Task Results**

Currently, Cuckoo only analyzes the tasks but does not do anything with it. In the future, it would be beneficial to analyze the report and based on whether a web server makes a significant change to the minion's system, have a mechanism to deduce whether a website is malicious or benign.

# Bibliography

- [1] HoneySpider Network 2. *NASK/CERT Polska and National Cyber Security Centre* (2013).
- [2] AKKAYA, D., AND THALGOTT, F. Honeypots in Network Security. Thesis, Linnaeus University: School of Computer Science, Physics and Mathematics, 2010.
- [3] BIDDISCOMBE, J., GEVECI, B., MARTIN, K., MORELAND, K., AND THOMPSON, D. Time Dependent Processing in a Parallel Pipeline Architecture. *IEEE transactions on visualization and computer graphics* 13, 6 (Jan. 2007), 1376–83.
- [4] CHEN, H.-P., AND ZHANG, C. A Fault Detection Mechanism for Service-Oriented Architecture Based on Queueing Theory. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)* (Oct. 2007), IEEE, pp. 1071–1076.
- [5] ERIC PETER, AND TODD SCHILLER. *A Practical Guide to Honeypots*, 2008.
- [6] FRANK BUSCHMANN, REGINE MEUNIER, HANS ROHNERT, PETER SOMMERLAND, AND MICHAEL STAL. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley Series in Software Design Patterns, 1996.
- [7] HURA, G. Client-server computing architecture: an efficient paradigm for project management. In *Proceedings for Operating Research and the Management Sciences* (1995), IEEE, pp. 146–152.
- [8] HUSSAIN, S., AND ABDUL QADIR, M. ACID: Adaptive, convergent, and intelligent fault monitoring for distributed systems. In *2008 4th International Conference on Emerging Technologies* (Oct. 2008), IEEE, pp. 126–131.
- [9] KATE MATSUDAIRA. Scalable Web Architecture and Distributed Systems. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson, Eds., vol. 2. ch. 1.
- [10] LEAVITT, N. Will NoSQL Databases Live Up to Their Promise? *Computer* 43, 2 (Feb. 2010), 12–14.
- [11] LUKE KANIES. Puppet. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson, Eds., vol. 2.
- [12] PIOTR KIJEWSKI, CAROL OVERES, AND ROGIER SPOOR. HoneySpider Network: Fighting Client-Side Threats. *25th Task Force for Computer Security Incident Response Teams Meeting (TF-CSIRT)* (2008), 1–47.
- [13] RADEK HES, RAMON STEENSON, AND CHRISTIAN SEIFERT. The Capture-HPC Client Architecture. *New Zealand Chapter: Capture-HPC Alliance* (2010), 1–8.

- [14] RAMON STEENSON, AND CHRISTIAN SEIFERT. Capture-HPC: The Capture Communication Protocol, 2007.
- [15] VALLEE, G., NAUGHTON, T., AND SCOTT, S. L. System management software for virtual environments. In *Proceedings of the 4th international conference on Computing frontiers - CF '07* (New York, New York, USA, May 2007), ACM Press, p. 153.
- [16] YANG, J., CAO, J., WU, W., AND TRAVERS, C. The notification based approach to implementing failure detectors in distributed systems. In *Proceedings of the 1st international conference on Scalable information systems - InfoScale '06* (New York, New York, USA, May 2006), ACM Press, pp. 14–es.

