HONEYBOT: A HONEYPOT FOR ROBOTIC SYSTEMS

A Thesis
Presented to
The Academic Faculty

By

Celine Irvene

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2017

HONEYBOT: A HONEYPOT FOR ROBOTIC SYSTEMS

Date Approved: December 8, 2017

If you want to go fast, go alone. If you want to go far, go together.

African Proverb

Won’t He do it. Will He won’t.

ACKNOWLEDGEMENTS

All glory be to God. This is for my mother and my father, for my family, my friends, and all those who supported me. Ya know what ¯_(ツ)_/¯. Shout out to those who didn’t support me too. Still made it!!! Two down, one to go.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Historically, robotics systems have not been built with an emphasis on security. Their main purpose has been to complete a specific objective, such as to deliver the correct dosage of a drug to a patient, perform a swarm algorithm, or safely and autonomously drive humans from point A to point B. As more and more robotic systems become remotely accessible through networks, such as the Internet, they are more vulnerable to various attackers than ever before. To investigate remote attacks on networked robotic systems HoneyPhy, a physics-aware honeypot framework, has been leveraged to create the HoneyBot. The HoneyBot is the first software hybrid interaction honeypot specifically designed for networked robotic systems. By simulating unsafe actions and physically performing safe actions on the HoneyBot the intent is to fool attackers into believing their exploits are successful, while logging all the communication to be used for attribution and threat model creation.

This thesis presents the HoneyBot and discusses a proof of concept implementation as well as the HoneyBot simulator. The HoneyBot prototype swaps between physical actuation and using prebuilt models of sensor behavior for simulation at runtime given user input commands.

CHAPTER 1
INTRODUCTION

A robot is a device or mechanism guided by automated controls that can do the work of a person or perform complicated repetitive tasks [1]. Robots are used for a variety of different reasons, the most prominent of them being the elimination of the use of human operators. There are three main reasons for this; the first is to save on labor, which ultimately reduces costs. For instance, in the car manufacturing industry robots on the plant floor cost about \$5.67 an hour over the course of their lifetime (including maintenance and energy costs), while the comparable human labor costs over \$40 an hour (including wages, pension, and health care) [2]. The second reason to eliminate human operators is because human interaction might be bad for the product. This is seen in the semiconductor manufacturing process, which consists of hundreds of steps. If at any point there is contamination (usually due to human error), the whole wafer could be ruined and the process must be restarted with a new one. The final, and arguably, most important reason for using robots is that the product or job may be too dangerous for humans. Imagine an encampment near a war zone, hours after a covert enemy attack has wiped out a squadron. The grounds are littered with corpses and debris and immediately after the assault not much can be done to rectify the situation. Rather than sending more troops to risk their lives, military robots can be sent to identify and collect the remains. Now imagine that same encampment if military robots had been used for battle instead of soldiers, the loss of life would have been minimized and even though there would have been equipment loss, the underlying intellectual property would still be available and the machinery rebuilt.

The current measures available for avoiding scenarios like those previously mentioned, though limited, are growing and being deployed each day. These measures include technologies such as drones and unmanned ground vehicles. To reap the benefits of these robots they must be autonomous, remotely controlled, or a combination of the two. Given the variability of most missions, it is usually more desirable to have remote access through a network. While remotely accessible robots allow for more convenience and easier operational use, it opens them up to similar vulnerabilities as the ones faced in traditional computer networks. A classic example of this is Stuxnet, commonly known as the world's first digital weapon. Stuxnet was a 500KB computer worm that infected the software of at least 14 industrial sites in Iran, including a uranium-enrichment plant [3]. It worked by compromising the Siemens Step7 software used to program the programmable logic controllers (PLCs) driving the Iranian centrifuges. Once Stuxnet had access to the PLCs, the worm's creators were then able to take control of the industrial systems and caused the fast-spinning centrifuges to tear themselves apart, unbeknownst to the plant operators. The Stuxnet attacks were speculated to have been done over the course of several months by manipulating process pressure and centrifuge rotor speeds in an attempt to damage them and delay the Iranian nuclear program [4]. Stuxnet, if nothing else, has demonstrated that injecting malicious code on cyber physical systems (CPSs), often embedded and real-time, is possible and that every networked or remotely accessible system is at risk.

A CPS attack that occurred more recently than Stuxnet took place at the University of Washington where security researchers were able to hack a teleoperated surgical robot [5]. The lack of trained surgeons has been a major obstacle preventing life-saving surgeries from being performed in many parts of the world, and one solution to this involves leveraging the Internet and robotics. Teleoperated surgical robots enable expert surgeons to be in one country or region controlling a robot that physically performs a surgery in another country or region. The sale of these types of medical robots and others has been increasing at a rate of 20 percent per year and is expected to keep growing [6]. The rapid rise in popularity of teleoperated surgical robots has drawn the attention of several security experts at the University of Washington, leading them to discover that malicious attackers can disrupt the behavior of telerobots during surgery or take them over completely. The attack was performed on Raven II, a robot designed to reduce the size of medical robots and to improve their durability for use in extreme environments. The robot is built on a Linux based computer running the Robot Operating System [7] and communicates via the Interoperable Telesurgery Protocol [8] over a public network with support for wireless and low link quality connections. This connectivity which is so helpful for saving lives is also what makes it vulnerable. ”Due to the open and uncontrollable nature of communication networks, it becomes easy for malicious entities to jam, disrupt, or take over the communication between a robot and a surgeon,” according to the researchers [6]. The attack’s only requirement was that the malicious user be connected to the same network as the control console of the telerobot. The researchers performed three kinds of attacks on an experimental operation where the operator must move blocks from one peg board to another. They then measured the rate of task completion and the operator reported difficulty while under attack. Of the three types of attacks performed, the most appalling was how they were able to hijack a connection over the Interoperable Telesurgery Protocol due to insecure sequence number processing. This allowed them to take control of the teleoperated procedure and inject packets that triggered the built-in automatic stop mechanism, resulting in a denial of service (DOS) attack on the robot [5].

The prevalence of robotics is growing in all facets of everyday life and robots are becoming a crucial part of the ecosystem. They are relied upon for military purposes on the war front, they are relied upon for assisting doctors in the healthcare industry, and even first responders and the police use them. If steps aren’t taken to secure robotic devices they will become serious safety threats.

In computer security, the first step to securing a resource is traditionally the development of a threat model. A threat model can help assess the probability and the potential harm, which can be useful in minimizing or eradicating the threat. Historically, security in robotics has not been an eminent concern, so to determine a valid threat model these systems must be studied and monitored to learn the scope of attacks they could face. This thesis asserts that this can be accomplished with a honeypot specially designed for robotic systems, called the HoneyBot. The HoneyBot is the first software hybrid interaction honeypot specifically designed for networked robotic systems. By simulating unsafe actions and physically performing safe actions on the HoneyBot the goal is to fool attackers into believing their exploits are successful, while logging all the communication to be used for attribution and threat model creation.

1.1 Primary Contributions and Thesis Organization

The primary contributions of this thesis are the following:

- An initial version of the HoneyBot is proposed, which leverages HoneyPhy, a physics-aware honeypot framework
- A software proof of concept HoneyBot simulator is presented in Python
- A hardware proof of concept is implemented for the HoneyBot using the GoPiGo 3 robotics platform
- Five device models are created for the sensors of the HoneyBot
- Results from a user experiment where participants control the HoneyBot are presented

The remainder of this thesis is presented as follows: Chapter 2 describes related honeypot works and CPS specific honeypots. It also details honeypot evasion techniques, alluding to how the HoneyBot avoids these techniques. Chapter 3 provides background information on robotic systems and describes the architecture of the HoneyBot. Chapter 4 describes experiments performed to prove feasibility and build the required sensor *Device Models*. Finally, Chapter 5 presents the conclusion and potential avenues for future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Honeypot Classification

Since their inception, honeypots have primarily focused on the traditional IT computing domain, seeking to monitor attackers that aim to compromise company and government workstations/servers. Several different classes of honeypots were created. Honeypots were classified based on their levels of interaction and simulation. In addition to these previously defined classes, previous work in the Communications Assurance and Performance (CAP) Group[9], defines a ”hybrid” interaction honeypot. This is a honeypot that dynamically swaps between being low-interaction and high-interaction given certain triggering conditions. Table 2.1 breaks down the differences between these classes of honeypots.

Table 2.1: Breakdown of key features for each interaction level honeypot

Levels of interaction	How does it work?	Ease of deployment	Risk	Detection
Low interaction	Simulates services and applications	Simple	Low risk - do not run in production system	Easier to detect
High interaction	Utilizes real OS and applications	Complex	High risk - runs in production system	Difficult to detect
Hybrid interaction	Dynamically switches between real system and simulation	Simple	Medium risk - runs within production system	Difficult to detect

2.2 Cyber-Physical Systems Honeypots

The first honeynet (a network of honeypots) for CPSs/SCADA (supervisory control and data acquisition) was created by Pothamsetty and Franz of the Cisco Infrastructure Assurance Group (CIAG) in 2004 [10]. The goal was to simulate a few popular PLC services to help researchers better understand the risks of exposed control system devices. This work has laid the foundation for many other CPS honeypots [9] [11] [12], none of which, though, are directly applicable to the robotics domain. With the prevalence of robotic systems on the rise, it is critical that advanced monitoring techniques, such as honeypots, be extended to defend them. Though the inspiration for this work comes from [9], and leverages the concept of hybrid interaction honeypot, the HoneyBot deviates in its robotic system specialization.

2.3 Honeypots and Evasion

Honeypot evasion is, as with most security problems, a cat and mouse game. There almost always exist configuration fingerprints for any honeypot, and they are corrected as attackers discover them and defenders must devise appropriate improvements. An early example is the high-interaction honeynet, Sebek [13], which was discovered and trivialized (exposed as a honeynet) by techniques described in [14]. The techniques revealed how to detect, circumvent, and blind Sebek. Consequently, the fingerprints Sebek exposed were remedied by leveraging a similar system called Qebek that utilized virtualized high-interaction systems [15]. For many non-high-interaction honeypots, evasion boils down to finding the edges of emulation for the presented services, but timing approaches have also been used [16]. For example, Kippo is a popular medium-interaction honeypot for the SSH (Secure Shell) service [17]. Kippo is easily detected by sending a number of carriage returns, and noting the output difference from production SSH servers [18]. While high-interaction honeypots are harder to detect, many rely on virtualization. Virtualization technologies usually leave their own fingerprints, such as device names, device driver names, file-system hallmarks, and loaded kernel modules [19]. Even though these fingerprints can be altered, there exists a rich set of techniques for detecting virtualization (and defeating detection attempts) from the malware-analysis field [20].

Other, more honeypot-technology agnostic, detection techniques have been proposed. Some of these techniques rely on the liability issues inherent in hosting deliberately compromised machines. A botnet architecture proposed in [21] leverages the honeypot owner's desire to restrict outgoing malicious traffic to authenticate new hosts before integrating them into the botnet. Specifically, the new host is directed to send apparently malicious traffic to an already compromised ”sensor”. Most honeypot systems will attempt to identify and block or modify this malicious traffic, so whether or not the sensor receives the traffic unaltered can be used to determine if the new host is genuine. This work was built upon [22], where multiple pieces of evidence can be formally combined to derive a metric of likelihood that a host is a honeypot. This evidence could be the virtualization status of the host, the diversity of software on the host, the level of activity of the host, or the difficulty in compromising the host. This newer technique is presented in the context of a botnet, but the generalized belief metric is equally applicable to any honeypot technology, depending on the evidence used. There are also techniques described in two more recent surveys [23] [24], which elaborate on the themes mentioned above. Themes such as finding edges of emulation, finding subtle discrepancies that indicate virtualization, or analyzing the results of communication with an already compromised sensor.

2.4 How the HoneyBot is different

The HoneyBot attempts to address the honeypot shortcomings stated in Section 2.3, specifically for robotic systems. Firstly, the robotic system exists and is in use, which remedies the traditional lack of context and provides convincing evidence correlated with a real system. The HoneyBot is not virtualized, but implemented on actual hardware. The robotic system fully implements the presented services, so all system responses are in line with the HoneyBot devices. The HoneyBot is the first software hybrid interaction honeypot specifically designed for networked robotic systems. By simulating unsafe actions and physically performing safe actions on the HoneyBot the goal is to fool attackers into believing their exploits are successful, while logging all the communication to be used for attribution and threat model creation.

CHAPTER 3
ROBOTICS AND HONEYBOT ARCHITECTURE

3.1 Robotics

The field of robotics is constantly changing, but the components that unite almost every class of robots are sensors, actuators, and controllers. Figure 3.1 shows a labeled diagram of the GoPiGo2 robot [25] denoting these components. Some background information on the basic building blocks of robotics is, required to describe how the HoneyBot presents a honeypot for a full robotic system and is presented in the next three subsections.

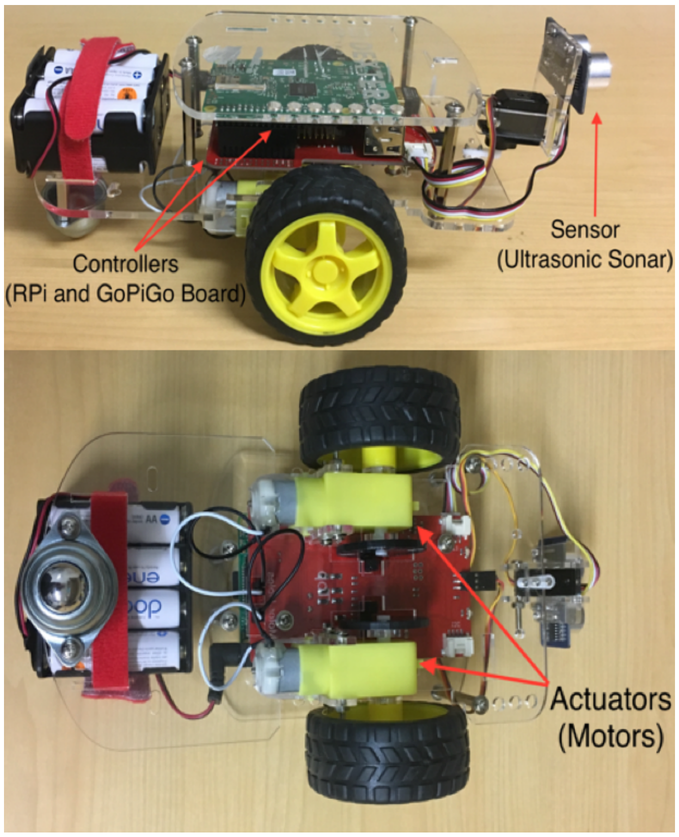


Figure 3.1: GoPiGo2 robot with labeled components

3.1.1 Sensors

Sensors are the robot's eyes and ears; they enable the robot to understand the world around it and judge several features of the environment. Some examples of common sensors on a robot include: cameras, thermometers, IR (infrared) and sonar, accelerometers, and magnetometers. The HoneyBot must be able to ”spoof”, through simulation, all sensor values outputted by the robot such that the attackers are incognizant of the change. To do this device models were developed that provide realistic system responses given an ”unsafe/indeterminate” input from an attacker. One such model that was developed is for the Grove SEN10737P Ultrasonic Sonar. This device model was crafted through lab experimentation and empirical observation; such that for a given speed and distance the model mimics the true response time of an ultrasonic pulse, and ”spoofs” the distance measured. This data is used to provide a reasonably timed response to the user such that there is no distinguishable difference when comparing the time taken for a simulated response versus an actual response. More details about experimentation and device modeling can be found in Sections 3.4.1 through 3.4.4.

In addition to spoofing individual sensor values, the HoneyBot must also spoof the correlation between sensors values. For instance, if the purpose of a certain robot is to navigate efficiently through an indoor-outdoor obstacle course, an attacker who sends commands to continually ram the robot into a wall in the outdoor portion of the course would not only expect to see the distance sensors reflecting these actions, but also the temperature sensor should reflect the outdoor temperature during a hot summer day. In this scenario, the HoneyBot would be triggered to perform simulations once the bad commands are received and would not actually allow the robot to crash into the wall, but instead output alternating distance sensor readings reflecting wall proximity and sensible temperature readings based on the known context of the robot's state. Given the sporadic nature of electronic components when subjected to conditions they are not built to support, one response (when destructive/unsafe commands aimed at destroying the robot are received) of the HoneyBot will be to disallow the occurrence of the actions and take down the connection to the attacker in a manner that suggests the robot has failed/died and was abruptly taken offline.

3.1.2 Actuators

The next component common to every class of robots is the actuators. Actuators enable the robot to modify the environment and move (or actuate). Some examples of these components include motors, speakers, and tools that go on the end of robotic arms, known as manipulators, like hammers, shovels, or even scalpels in the case of surgical robots. Since the proof of concept HoneyBot implementation lives in a robot that operates in networked environments and users/attackers have no physical or visual access to it, this work does not focus on modeling the underlying control system of the actuators. Instead, the emphasize is on accurately simulating the timing of system responses and how user commands will change the state of the overarching system. In addition to this, realistic device fingerprints and responses are generated and sent back to remote users over the network. To do this, the HoneyBot must know what commands are safe and can be performed, and what commands are unsafe and must be simulated. Determining the ”safety” of user commands is a hard problem to solve, and many researchers have worked on formal methods for verifying the safe navigation of ground robots [26] [27] [28] [29]. Though it is not the focus of this work, these works could be leveraged and incorporated into the HoneyBot as the *Input Verification* module is built for plug and play. As shown in Figure 3.2 all the commands received by the robotic system are passed through the *Input Verification* module to determine their fitness to be performed. Given the command and the current system state, the module must output whether the action is safe or unsafe. If deemed safe, the HoneyBot will allow the command to be performed. Otherwise, the command is not sent to the actuators but instead is simulated.

3.1.3 Controller

The third, and arguably most important, component of a robot is the control system or controller. The controller, also known as the brain of the robot, enables the robot to parse commands, send signals to different devices, and communicate with other robots as well as with the user. The HoneyBot is software that will live in the robot’s controller so that it can easily access all data commands and signals to and from the robot’s brain allowing it to make the necessary decisions.

Figure 3.2 shows the flow of the HoneyBot system architecture, a user/attacker remotely connects to the networked robotic system through the Internet and can send commands. All the commands received by the robot will be logged and passed to the *Input Verification* module, which as stated earlier is flexible in structure and can be very comprehensive in the evaluation of commands or relaxed depending on application needs. If the *Input Verification* module deems a command safe, the action will be performed as usual and the system response will be returned to the user/attacker. If, on the other hand, the *Input Verification* module deems a command unsafe the command will be simulated in real time and the ”spoofed” system response will be generated and returned.

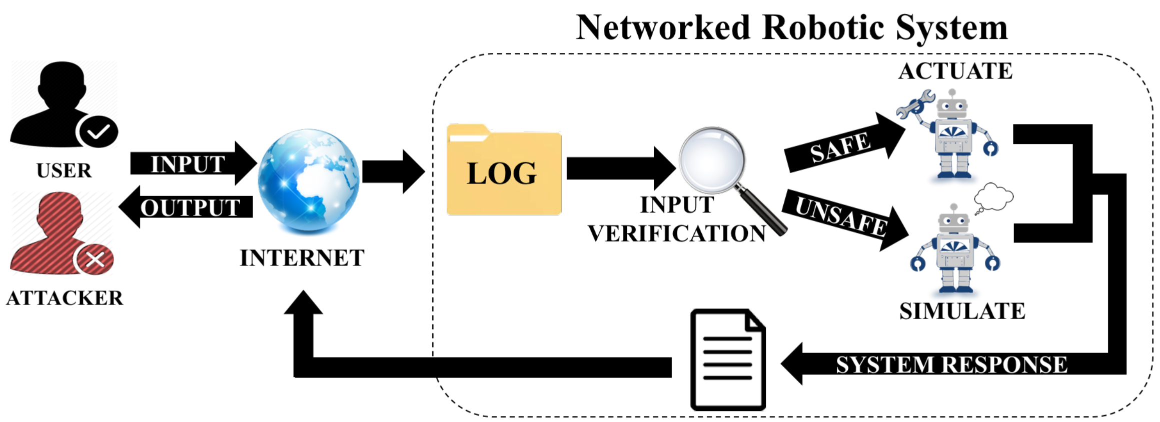


Figure 3.2: HoneyBot system architecture

3.2 Honeypot for Robotic Systems

Most existing CPS honeypots neglect certain aspects of these systems that can alert an attacker to the nature of the honeypot, namely the physics of the devices that interact with the process [9]. That is why the HoneyBot is based on HoneyPhy. HoneyPhy is a generic physics-aware honeypot framework that accurately models software and protocol fingerprints which are then used to simulate the CPS and fool attackers who access the honeypot [9]. The HoneyBot then applies this generic framework, and tailors it to the specifics of a robotics system. The HoneyPhy framework is composed of three main modules: the *Internet Interface*, the *Process Model*, and the *Device Model* modules. As applied to the HoneyBot, the *Internet Interface Module* is used for opening ports or interfaces on the robotic system so that it can connect to a network. It can also be used to support multiple connections to other networked robots and verify that state ensuring commands are always received by the correct robot at the correct port/address. In other words, the *Internet Interface* is the user facing interface, the front end that the attacker can see. The *Process Model* is triggered by an ”unsafe” or ”indeterminate” command, and the action will be simulated in real time by querying the appropriate device model rather than sent to be deterministically performed on the robot. The *Device Model* is different from the other modules in that it contains a model representative of each device found within a robot. These models are built using real data gathered from a given device. Several proof of concept device models were built, details of which can be found in Section 3.4.

Device models are used for ”spoofing” response times and data from sensors when the HoneyBot receives malicious commands. The details of the experimental set up and infrared distance sensor model building can be found in Section 3.4.3. One example of a possible model to include amongst the models in a robotic systems’ *Device Model* could be of the robot’s motor. When the HoneyBot is sent a command to normally power the motor or any otherwise safe/benign command, the HoneyBot will allow the action to be performed. However, when the HoneyBot is sent a command to overpower the robot’s motor, the *Process Model* will query the motor’s *Device Model* and simulate the overpowering of the motor. The HoneyBot then changes the system state and responds to the attacker with data reflecting the sudden change in power to the motors. In general, the procedure for calling the *Process Model* will entail simulating an incoming malicious command and responding with the data obtained through simulation. Due to the seamless switches between allowing the robot to physically perform ”safe” commands and simulating ”unsafe” or ”indeterminate” commands, the HoneyBot is classified as a hybrid interaction honeypot.

3.2.1 Sensor Correlation

Sensor correlation is the data consistency across a robotic system that keeps attackers’ believing they are interacting and controlling a real system. For any machine that performs tasks and reports back to users, sensors (in addition to supplying critical information for understanding an environment) provide vital insight to the equipment's operational status. Moreover, they can relay information about the task as well, like whether it was performed to completion or not. For instance, with the HoneyBot the battery is a tell-tale sign of how much work the robot has done as well as how much more work it can do on that charge. Sensors can also conflict with each other. Consider an IR proximity sensor; it is a very effective means of measuring distance, but it is often susceptible to interference from outside light sources and changes in material reflectivity. Now consider ultrasonic sonar; it is also efficient at range detection (and for longer distances), but can be fooled by textures and echoes. Individually, if faced with either of their weaknesses these sensors will fail to perform accurately, but when used in conjunction they prove almost infallible.

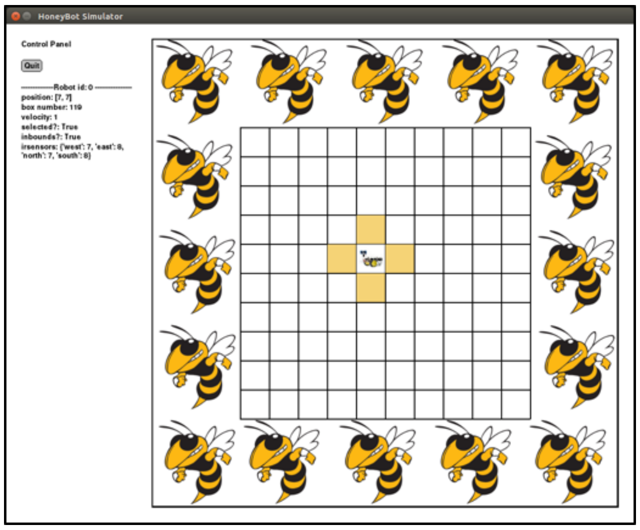
Formally, let S represent the set of all n sensors on a specific robot, with some sensors being correlated and others completely independent. An intelligent attacker wishing to determine whether his malicious commands are being sent to a honeypot or to a real system can first build a ground truth of the sensor correlation behavior by collecting data during "normal" operation and performing pairwise Pearson correlation calculations on the sensor data, $r_{xy} = \frac{cov(S_x,S_y)}{\sigma_{Sx}\sigma_{Sy}}$ to build an n x n correlation matrix, R_{true}. Now, as he begins to send malicious commands to the robot, he can continually monitor the sensor data streams, calculating new correlation matrices R_{test}, and monitoring for changes in the correlation behavior. Mathematically, to make a decision D_i deciding if sensor S_i is real or simulated, the attacker first chooses some distance metric d, such as the L2 norm, and some threshold ϵ. The attacker then determines the L2 norm of the difference between row i of R_{test} with row i of R_{true}. If the result is greater than the threshold, this tells the attacker that the sensor correlation has changed and he is most likely in a honeypot, as illustrated in Equation 3.1.

$$D_i = \begin{cases} real & d(R_{true}[i], R_{test}[i]) \leq \epsilon \\ simulated & else \end{cases} \tag{3.1}$$

This is why sensor correlation within the HoneyBot is so important. Under normal operation many sensors in a system will be related and, thus, under simulation these relationships must still hold and the sensor values must dynamically update with each user command. In order to accomplish this, each device model for the HoneyBot sensors was built using physical device properties and real device data. Device modeling causes simulated responses to mirror true operation in an effort to avert attacker detection for as long as possible.

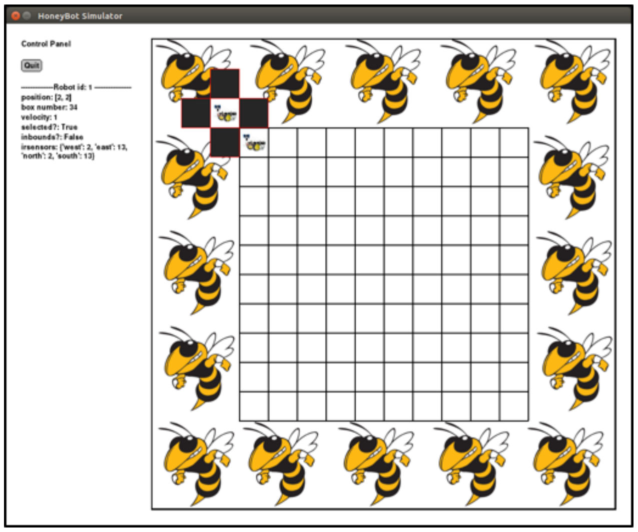
3.3 Proof of Concept HoneyBot Simulator

As a proof of concept model for the HoneyBot, a simulator was developed. The HoneyBot simulator is a GUI (graphical user interface) tool that provides a graphical and interactive resource for users to envision an attacker scenario in a networked robotic system (where the HoneyBot has been implemented). Screenshots of the simulator can be seen in Figure 3.3.



```
------------Robot id: 0 ------------
position: [7, 7]
box number: 119
velocity: 1
selected?: True
inbounds?: True
irsensors: {'west': 7, 'east': 8,
'north': 7, 'south': 8}
```

(a)



```
------------Robot id: 1 ------------
position: [2, 2]
box number: 34
velocity: 1
selected?: True
inbounds?: False
irsensors: {'west': 2, 'east': 13,
'north': 2, 'south': 13}
```

(b)

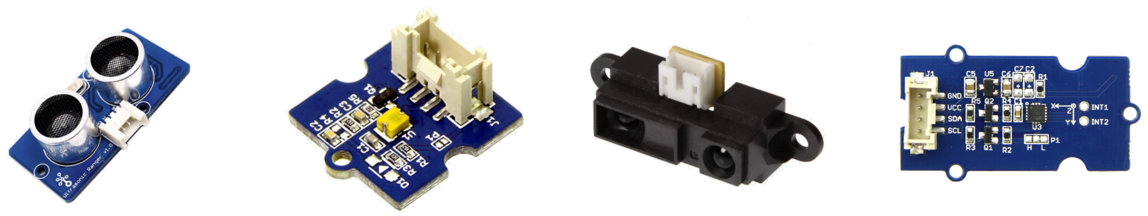
Figure 3.3: Screenshots of the HoneyBot simulator (a) HoneyBot Simulator with single robot inbounds (b) HoneyBot Simulator with single robot out of bounds. True position shown inbounds.

The simulator displays a control panel to the left of a 16 x 16 bounded area, where the center 10 x 10 grid is the legal boundary for the robots to roam and perform tasks. Normal and benign activity can be performed within the 10 x 10 grid; this is representative of the traditional user privilege zone. Outside of the legal boundary is the ”honey”, an extra boundary which is representative of resources that are attractive to an attacker, like open ports or unpatched software in a traditional computer network. In the simulator, a robot is represented by a GoPiGo2 image and has the following attributes: unique ID, velocity, position, box number, IR sensor values, legal moves, a value corresponding to whether the robot is within the legal bounds or not, and a value denoting if it is currently selected. *Position* is the robot's location in coordinate form and box number is the location in terms of the number of boxes from the first legal box (each of these location indicators can be derived from each other). The velocity is the number of boxes a robot can move at a time and the default value is 1. The IR sensor readings indicate how far in boxes the robot is from the legal boundary to the north, south, east, and west at any given moment. Given its velocity, a robot will have *n* number of possible next boxes that it can move to and this knowledge is always maintained by the bot. Two Boolean values tell the robot whether it is within the legal boundaries of the grid and if it is currently selected. Only one robot on the grid can move at a time and that robot must be ”selected”. This is indicated in the GUI by highlighting that robot. There is one attribute of the robot that is not exposed to the user, and this is the robot's true position. The true position is the system administrators' way of keeping tabs on the actual location of the robot. True position is the only sensor that will not ”lie”. In the simulator, when a robot is out of bounds all the information sent back to the user reflects this, from the IR sensor readings to the position updates, and even to the next moves it can make. This falsified information, shown in the Control Panel, is what the user has access to, while the true position, as represented in Figure 3.3b where the robot is still inbounds, is only known to the system administrator. In this way, the attacker will be unaware he/she is being monitored and will continue performing the malicious actions. The malicious actions, in the case of a ground robot, could include the running of the robot into the edge of the true boundary to cause physical damage or robot-robot collisions.

3.4 Proof of Concept Sensor Device Modeling

For sensor model development used to simulate the sensor behavior, a combination of techniques were employed, including experimentation and physical process modeling. The idea is that the models built would be queried at runtime to generate ”spoofed” responses which get sent back to attackers when they perform malicious or otherwise unsafe actions. Suppose that the HoneyBot was implemented in a military drone used for finding Improvised Explosive Devices (IEDs) and that it received a command directing it to go through a no-fly zone. Clearly, there is something suspicious so the *Input Verification* module will flag the action as unsafe. Then, the drone queries its GPS device model and sends back false coordinates, while maintaining its position in an unrestricted area, but leading the user to believe it is elsewhere. Device models must not only provide realistic state-aware data, but they must also reflect the correlation between sensors. For example, the distance sensors must corroborate the reported velocity data, and the velocity data must be in line with the encoder readings.

To create the most convincing HoneyBot possible, the sensor device models were built to closely mirror reality. To do this this a GrovePi (a circuit board for connecting various sensors to the GoPiGo2 robot) [30] was purchased as well as several Grove sensors from Seeed Studio. The GrovePi costs about $40 and the Grove sensors typically cost no more than $20 each. Device models were built for for the GrovePi SEN10737P Ultrasonic Sonar, the Sharp GP2Y0A21YK IR proximity sensor, STMicroelectronics LSM303D 6-Axis Accelerometer & Compass, and the Sensolute MVS0608.02 Collision sensor. Each of these sensors can be seen in Figure 3.4 and the rest of this chapter discusses them in detail.



(a) (b) (c) (d)

Figure 3.4: Images of the sensors (a) Ultrasonic sonar (b) Collision sensor (c) Infrared distance sensor (d) Accelerometer.

3.4.1 Ultrasonic Sonar

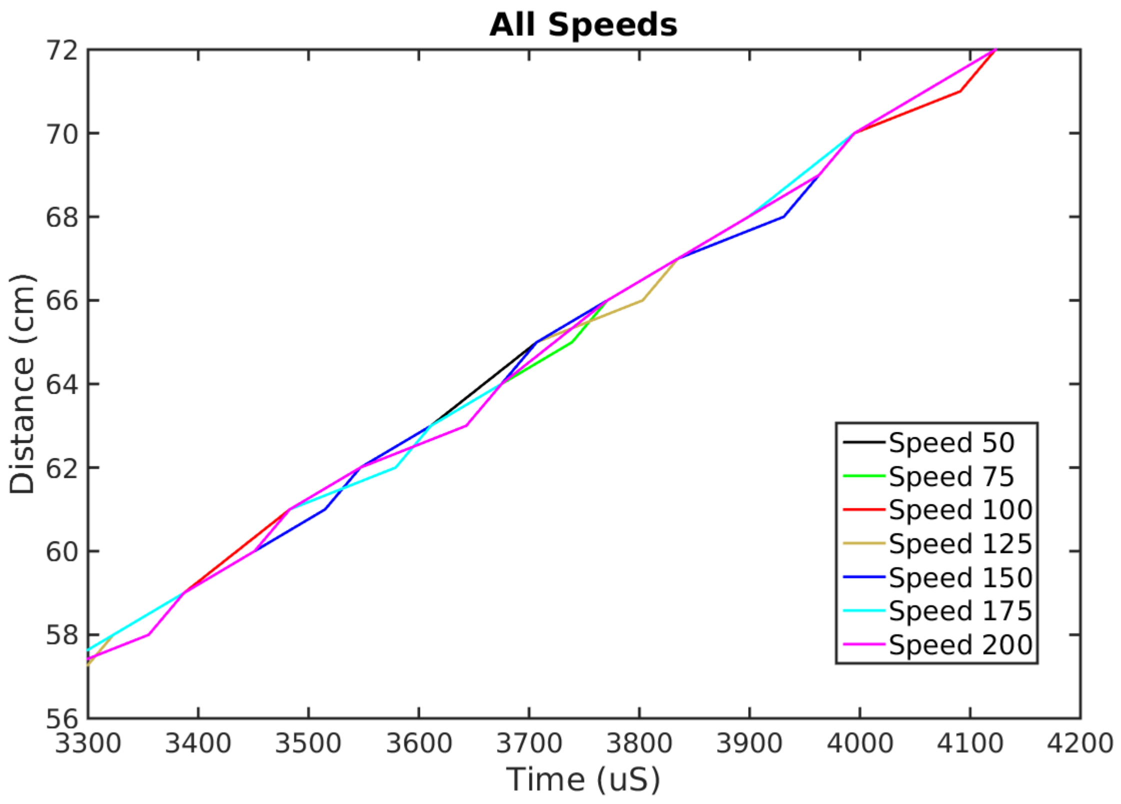


Figure 3.5: Zoomed in distance vs time for all 7 speeds on one plot.

The GrovePi SEN10737P Ultrasonic Sonar (shown in Figure 3.4a) is a non-contact medium range distance measurement module which works at 42KHz. This sonar is rated for measuring objects between 3cm and 400cm, but we found that it was only accurate up to approximately 200cm. This lack of reliability is likely due to its inexpensive manufacturing, as it was not built for high fidelity sensing. An experimental test environment was set up in a laboratory to measure the time taken for the ultrasonic sensor attached to the HoneyBot driving at different speeds towards a static target. The lab set up consisted of placing the robot 183cm away from a target and then driving it towards the target while taking distance and ultrasonic pulse time measurements. Ten trials per each of the seven chosen speeds (50 to 200 by increments of 25) of the robot were performed, 70 measurements in total. The speed on the GoPiGo2 robotic platform is unit-less and represents a percentage of the full power the motors can get. For example, 0 is for no movement at all (the motors are stopped) and 255 is for setting the duty cycle at 100%. The objective was to succinctly determine the time taken to perform an ultrasonic pulse for a specific speed and distance from a target. The times were all recorded in a dataframe along with the corresponding distance and speed and are queried at runtime when a malicious command is given and the HoneyBot must simulate a sonar reading. We created plots from seven lab experiments (where the ten trials have been averaged) that were run in order to map ultrasonic sensor obtained distances to the time taken for the ultrasonic pulse to be received as the HoneyBot approached a target at various speeds. Figure 3.5 shows the results of each speed plotted together. From Figure 3.5 (and underlying the sonar theory) we determined that at these low speeds and short distances the robot speed does not influence the time for an ultrasonic pulse to occur.

3.4.2 Collision Sensor

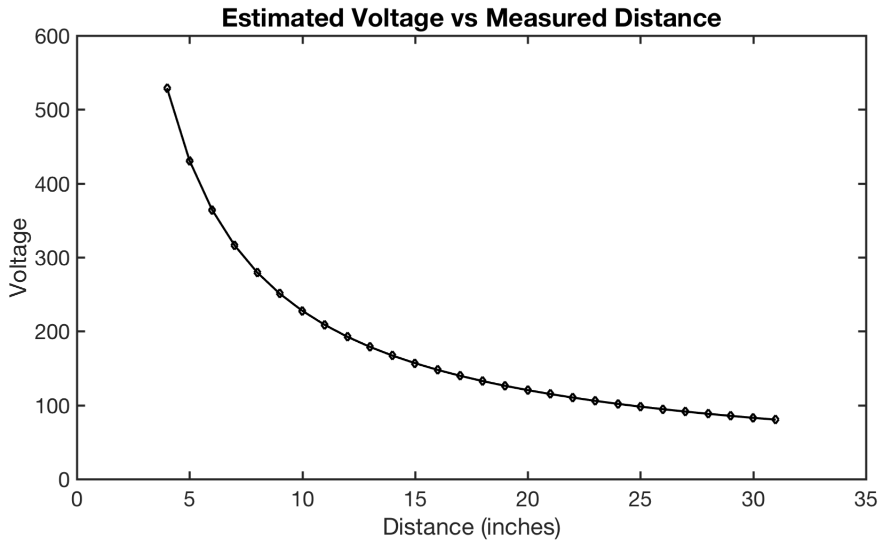
The Sensolute MVS0608.02 Collision sensor (shown in Figure 3.4b) is a micro vibration sensor that records motion and vibrations through a gold-plated moving micro-ball. Due to its small size, it is perfect for mounting on the HoneyBot and detecting obstacle collisions. The digital sensor outputs a high signal when a collision is detected and a low signal if no collision is detected. To develop the device model for simulating the collision sensor, the sensors sensitivity to collisions had to first be tuned to ignore the rumblings of the robotic wheels and to only identify major crashes as triggers. The sensitivity was tuned by introducing delay to the querying process and by verifying the collision status after the delay. For instance, instead of querying the device every quarter of a second we might query every half second. If the sensor reads high (implying there was a collision) then a delay of approximately 50ms was taken and the sensor queried again. If the sensor still read high, then it was concluded that a collision occurred. Due to the binary nature of the device, spoofing the output was as simple as returning a trigger event to the user when a simulated crash occurred.

The collision sensor device model was built and tested through a series of trials where the HoneyBot was placed on the ground varying distances from an obstacle and driven towards it. Each time a false positive trigger event occurred the delay was modified. It was eventually empirically determined that a delay of 50ms produced the most accurate results on the ground surface. The Sensolute MVS0608.02 Collision sensor costs about $10 and at this price point reliability tends to be inconsistent. It was found that the delay factor had to be tuned for different ground surfaces, but once properly calibrated has 90% accuracy.

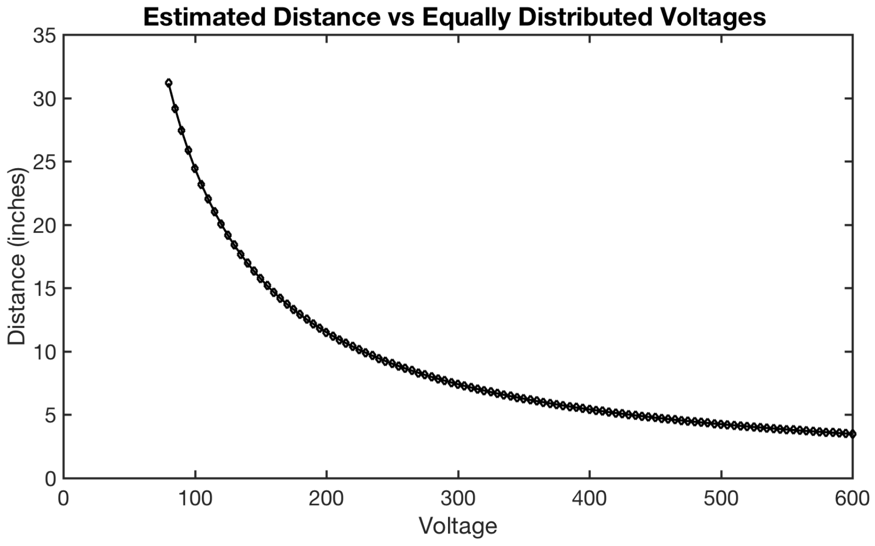
The key to simulating a collision is in the corroboration of sensors. The collision sensor device can easily be made to trigger a collision event, but it must first read from the distance sensors that it has closed in on an obstacle/barrier and they must reflect this by outputting a ”spoofed” position.

3.4.3 Infrared Distance Sensor

The Sharp GP2Y0A21YK IR proximity sensor (shown in Figure 3.4c) is a general-purpose short range distance measuring sensor. This sensor is rated for measuring distances between 10cm and 80cm. It works by using triangulation and a small linear CCD (charge-coupled device) array to output an analog voltage, which can be used to compute the distance of objects in its field of view. The measuring process starts with a pulse of IR light being emitted from the emitter, which travels out and either hits an object or does not. If there is no object, the light is never reflected and the voltage reading is low. If the light reflects off an object, it returns to the detector and creates a triangle between the point of reflection, the emitter, and the detector. Because of this triangulation the outputted analog values do not have a linear relationship to the distance from the object. Given this, a common method of determining distance is to build a plot of voltage outputted from the sensor versus distance (measured manually) and derive a function from the fit of the line. Equation 3.2 was derived from data collected by a user who performed the experiment described above, plotted, and fit a line [31]. This user’s data is verified by the Sharp datasheet [32] and thus Equation 3.2 is accepted as valid. In Figure 3.6a, we can see a plot created from Equation 3.2. Equation 3.3 was computed by simply solving Equation 3.2 for distance. A plot of Equation 3.3, where a vector of voltages equally distributed from 80 to 600 by steps of 5, can be seen in Figure 3.6b. Given these two equations and an understanding that the voltage increases as objects become closer, ”spoofing” the IR distance sensor was fairly straightforward. When a malicious navigation command is received by the HoneyBot, both the IR and the ultrasonic sonar device models will be queried and provided with a precalculated distance. The IR device model will use the precalculated distance in Equation 3.2 and return the voltage in the system response. The precalculated distance is determined by the HoneyBot through position estimation based on the malicious command given by the user and knowledge of the true and ”spoofed” state of the robot.



(a)



(b)

Figure 3.6: (a) Plot of the estimated voltage vs measured distance from the experimentally determined Equation 3.2. (b) Plot of the estimated distance vs equally distributed voltages from Equation 3.3.

$$observedVoltage = 1893.9 * distance^{-0.92} \tag{3.2}$$

$$distance = 3650.4/observedVoltage^{1.087} \tag{3.3}$$

3.4.4 Accelerometer

The STMicroelectronics LSM303D 6-Axis Accelerometer & Compass (shown in Figure 3.4d) is a 3-axis accelerometer combined with a 3-axis magnetometer (or compass). An accelerometer measures the acceleration of an object relative to inertia or free-fall, not an increase in speed like some might assume. This means the accelerometer will provide readings even when an object is static due to the gravitational pull of the Earth. One useful application of this is that in free fall the sensor will measure 0, highly sensitive accelerometers are used in the automotive industry for instantaneous airbag deployment in the event of an accident [33]. Since the proof of concept HoneyBot is a ground robot (moving in the X-Y plane) and the 3-axis accelerometer is not highly sensitive it does not provide much interesting data so no model was developed for it. For other types of robots, such as quadcopters, that have the ability to rotate on different axes and change orientation it may prove more useful. The compass, on the other hand, is much more useful for ground navigation as the heading (route course) informs the operator of the direction of travel. For the proof of concept operation of the HoneyBot there are four user accessible navigation commands; forward, backward, left, and right. As long as it is powered on, the HoneyBot will have a heading and to convince users of normal operation when commands are actually simulated, this heading must always correspond with other sensors and reflect updates in user commands. When building this model the two most important things were:

- Preserving the true heading of the HoneyBot, which is the real heading provided by the compass
- Maintaining the spoofed heading by referencing the true heading the first time a navigation

command is simulated and updating accordingly and from then on referencing the spoofed heading and updating it

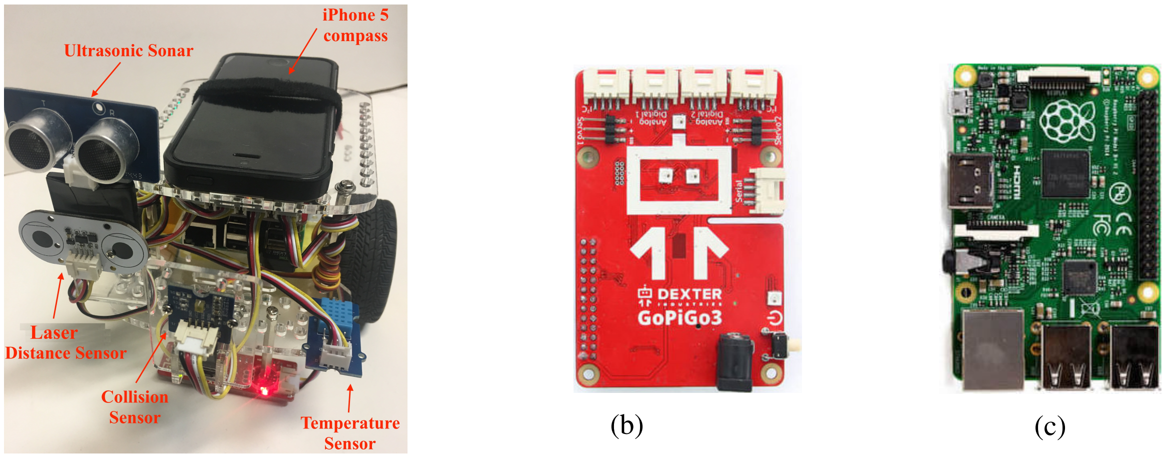
To accurately determine the spoofed heading after the initial deviation from the true heading 20 trials of a lab experiment were run. For the experimental setup the HoneyBot was placed on the ground at a denoted marker. A hardware compass was then used to measure the heading of the robot's direction. Then a Python script was executed which made the robot drive forward for 5 seconds, make a turn (left or right), and then continue driving in that direction for 5 more seconds before coming to a stop. Once the maneuvering finished the program measured the robot's new heading. The heading data from each of the trials was compiled into a CSV and used to determined the average degree of rotation for performing left and right turns. When queried for the spoofed heading, given the knowledge of the direction of turn, the compass *Device Model* will use the average degree of rotation plus some random noise to simulate the new HoneyBot heading.

CHAPTER 4
HONEYBOT EXPERIMENTATION AND EVALUATION

In order to prove the feasibility of the HoneyBot framework and architecture as it applies to real robotic systems, a proof of concept was constructed and evaluated through user experimentation. The design details of this proof of concept HoneyBot and experimentation are described in this chapter.

4.1 Proof of Concept HoneyBot

The GoPiGo 3 [25], shown in Figure 4.1a, was the chosen robotic system for the proof of concept HoneyBot. This platform was selected because of the ease of programming, through its support of the Python programming language, and the many I/O interfaces for attaching various robotic sensors. In addition to this the GoPiGo 3 was selected over the GoPiGo 2, used for initial model development, because of its magnetic encoders which ensure accurate robot control and its redesigned power management system which gives it longer battery life. These upgrades were crucial to performing the evaluation described in Section 4.2. The GoPiGo 3 Robot Car is a ground robot that consists of six major components: a GoPiGo 3 circuit board, a Raspberry Pi 3 [34], two motors, two wheels, various sensors, and a battery pack. The GoPiGo 3 circuit board, shown in Figure 4.1b, can be considered the secondary controller of the GoPiGo 3 Robot Car. It connects to the header pins of the Raspberry Pi 3, shown in Figure 4.1c, and receives motor control commands as well as provides status updates about the various connected sensors. The Raspberry Pi 3 is the main controller of the robot and can be accessed via direct connection (through its HDMI port), SSH, or VNC. The Raspberry Pi 3 runs the Raspbian OS, a version of Linux created especially for single board computers.



(a) (b) (c)

Figure 4.1: Images of the (a) fully outfitted GoPiGo3 Robot Car (b) GoPiGo3 Circuit Board (c) and Raspberry Pi 3.

4.1.1 HoneyBot Software

The HoneyBot software was written in Python 2.7 and is made up of three main modules:

- Robot Web Server: The robot web server is essentially the *Internet Interface Module* from the HoneyPhy framework and serves to communicate and transport commands from the front end (webpage) to the robot's actual hardware. The server was written using the Tornado web framework [35]. The web server is the process that is called to spin up every other module. When executed the web server instantiates a robot object (the robot controller), a HoneyBot object (the honeybot module), serves up the HoneyBot login page, and facilitates all web requests from clients through web sockets. The HoneyBot login page was used to safeguard the robot experimentation and evaluation process by defending access to the robots hardware with a rotating pairs of usernames and passwords. Before anyone could access the robot experiment website they had to enter a correct username/password pair and each set of credentials could only be used once before being invalidated, like a nonce.

- Robot Controller: The robot controller can be considered the *Process Model* from the HoneyPhy framework as it receives commands from the robot web server and translates them to navigational commands for the robot to perform or simulate. For instance, if the user clicks the right arrow key this is transported over a web socket from the client web page to the Tornado web server backend. The backend makes a call to the robot controller object which converts it to a navigation command and passes that to the Input Verification Module along with the robots' current status. The Input Verification Module then determines whether or not the command is safe to perform and if it is it gets sent to the robot's motors. If unsafe the honeybot module queries the sensor *Device Models* and spoofed data is returned. A code snippet defining this process is shown in Figure 4.2.

```python
def actuateRobot( self, direction, xLoc, yLoc):
  # Input verification stage
  safe = honeybot.InputVerificationModule().verify(direction,
 self.currStatus)

  if safe:
    # Action is safe - actuate
    gopigo.move()

    # Update status variables for GUI with real  data
    sysStatus = honeybot.trueStatusUpdate()
  else:
    # Action is unsafe - simulate
    gopigo.stop()

    logger.info("HoneyBot in danger zone")

    # Update status variables for GUI with spoofed data
    sysStatus = honeybot.simulateStatusUpdate(direction, xLoc,
 yLoc)
```

Figure 4.2: Simplified code snippet from Robot Controller.

- HoneyBot Module: The honeybot module is responsible for running a background process that constantly queries the robot for true sensor data. If the Input Verification Module detects an unsafe command the robot controller will call the honeybot modules' simulateStatusUpdate method and each of the robot's sensor *Device Models* will be queried for simulated data. The simulated data was collected through empirical observation and is described in detail in

Section 4.2.1.

4.1.2 HoneyBot Sensors

The proof of concept HoneyBot had five sensors, shown in Figure 4.3: a Sensolute MVS0608.02 Collision Sensor, an iPhone 5 Compass, a GrovePi SEN10737P Ultrasonic Sonar, a Dexter Industries Laser Distance Sensor, and an Aosong DHT11 Temperature Sensor. These sensors were chosen because of their significance to real-world ground robot applications. A collision sensor can be crucial to the well being of navigational robots, that are autonomous or remotely controlled, as they are the first line of defense for detecting and preventing costly damage to robotic end-effectors due to robot crashes [36]. A collision sensor on a deployed autonomous robotic system can be used to report damage to relevant parties who may be physically distant. In order to know where to dispatch rescue teams, monitoring parties need to know the robots' location. This is where the compass, laser distance, and ultrasonic sonar sensors come in to play. While they don't provide absolute location, like a GPS would, in many indoor or well-defined environments they are equally useful.

An iPhone 5 was used as the compass for the proof of concept HoneyBot because it provided more accurate headings than the STMicroelectronics LSM303D 6-Axis Accelerometer & Compass. The small magnetometer in the accelerometer could not overcome the interference from the many electrical components on the GoPiGo 3 and produced inaccurate data. The iPhone 5 has much better internal component shielding and did not suffer from interference when placed near the robot. An iOS mobile application, called RoboCompass, was written in the Swift programming language and downloaded to the phone. The RoboCompass App, source code can be found in Appendix A, sends compass readings in IP dataframes to the HoneyBot web server every time the compass reading changes (every time the robot moves).

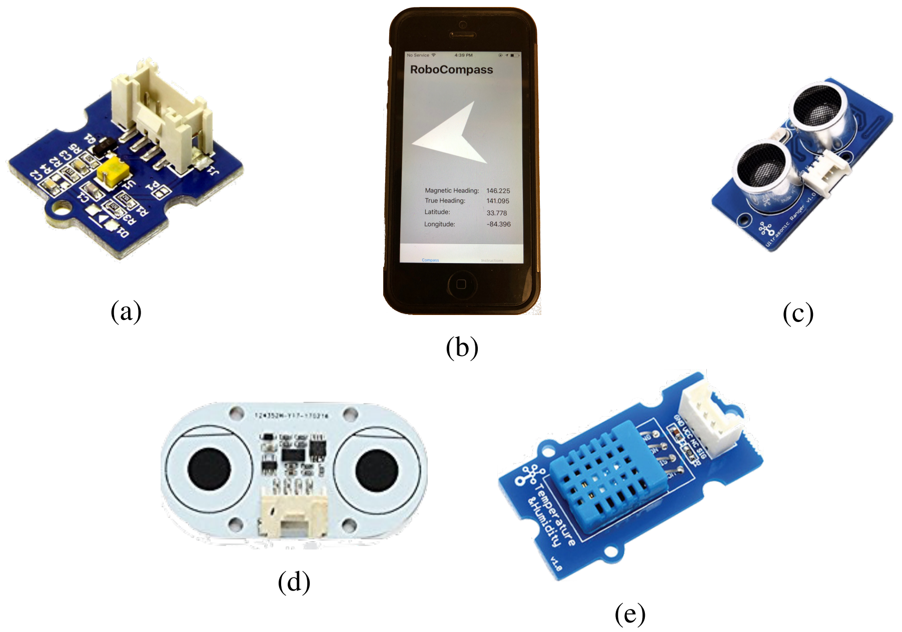


Figure 4.3: Images of the HoneyBot sensors (a) Collision sensor (b) iPhone 5 Compass (c) Ultrasonic sonar (d) Laser Distance sensor (e) and a Temperature sensor.

4.2 HoneyBot Experimental Design

Since the HoneyBot proof of concept was built on a ground robot, the best form of evaluation was determined to be a navigational task. To support this, an evaluation arena was built in the form of a 10 x 12 foot maze (shown in Figure 4.4) and participants (with no prior knowledge of the research) were recruited over the course of one week to remotely navigate the HoneyBot through it. Before beginning this study IRB approval was requested from the Georgia Tech Office of Research Integrity Assurance and the experiment protocol was designed.

The ”HoneyMaze” was constructed from approximately six 2 x 4 foot pegboards (used for the base or ground surface) and several hundred 1/2 x 48 inch wooden round dowels. The wooden dowels were cut, using circular saw equipment from the Georgia Tech ECE Senior Design Lab, into 6 inch pegs. These 6 inch pegs were then strategically ”nailed” into the pegboard base, one peg per every 3 peg holes, in the design of the preselected maze. After the pegs were secured in place rolls of 48 inch x 25 ft reflective insulation were cut into 7 inch tall strips and hot glued to one side of the pegs to create barrier walls. Reflective insulation was used as the wall material because a positive correlation was identified between the robot's distance sensor accuracy and the reflectivity of the surfaces measured against.

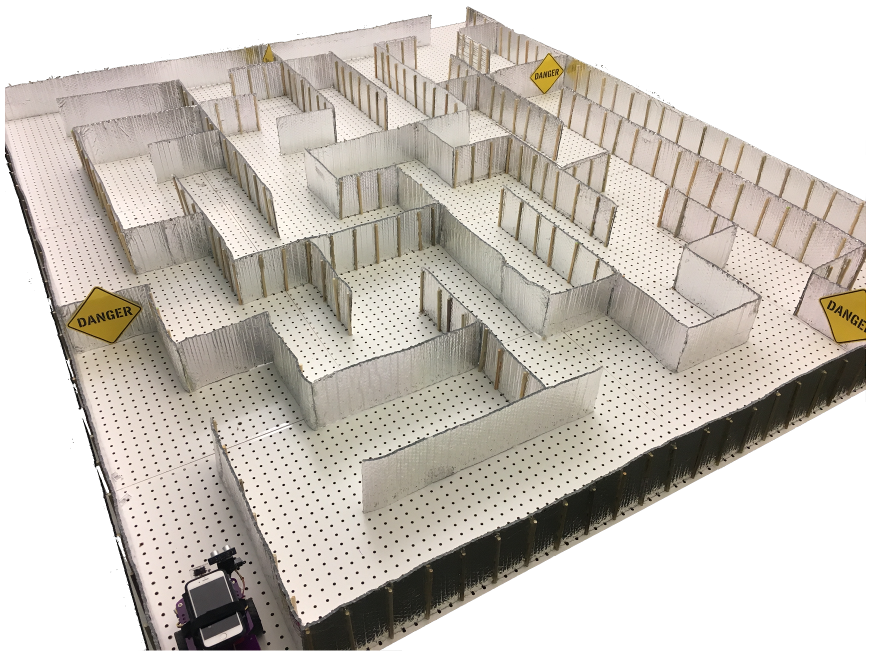
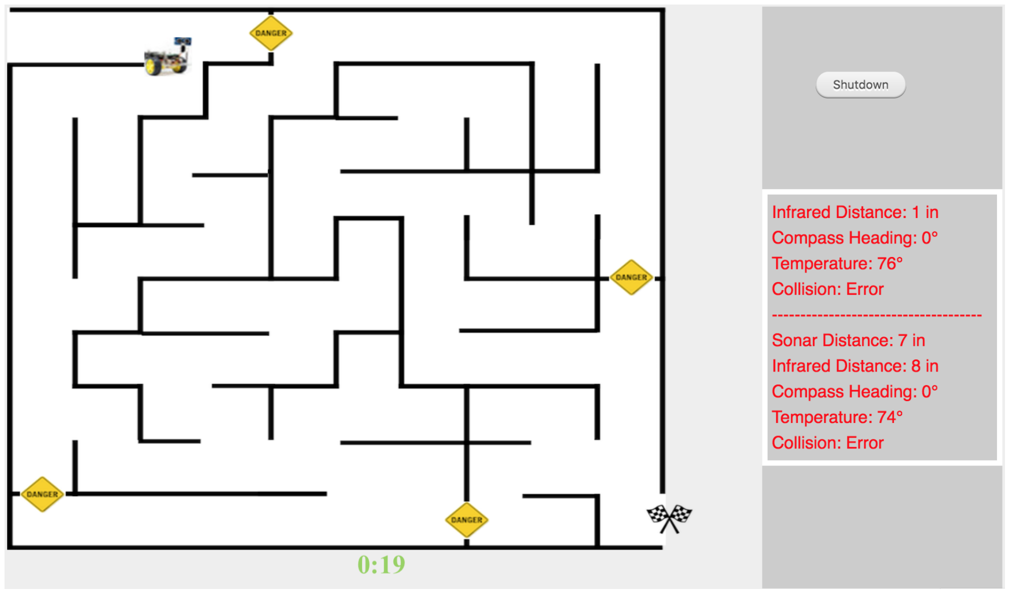


Figure 4.4: HoneyMaze with danger signs throughout.

The experiment required individuals to read instructions on how to navigate a robot through an online maze and then access the robot through a web interface. The participants were told the online robot corresponded to a real robot who at their every arrow keystroke would actuate through a life sized maze identical to the one on their screen. The exact instructions given to the research subjects can be found in Appendix B. Subjects were told that their mission was to navigate the robot through the maze fast as possible using only the online GUI maze and live sensor values from the robot displayed on screen. They were informed that the research objective was to determine the optimal constraint profile for the best performance and efficiency of the remotely controlled robot. A screenshot of the website is shown in Figure 4.5. At the bottom of the figure is a timer, participants were given 75 seconds to preview the maze and plan a route, then 60 seconds to actually navigate the robot to the finish flags. The ”constraints” they were told was that the robot moves very slow and they should plan their routes wisely, making sure to consider all possible options. The research subjects were otherwise given no specific guidance concerning the danger signs and when asked about them experiment proctors only responded with ”the meaning of the danger sign is up for interpretation, consider all possible options”. To assist with the participant recruitment process and as an added incentive for subjects to strive to complete the maze quickly, they were promised \$5 for participating and \$10 if they completed the maze before their 60 seconds ran out.



Infrared Distance: 1 in
Compass Heading: 0°
Temperature: 76°
Collision: Error

Sonar Distance: 7 in
Infrared Distance: 8 in
Compass Heading: 0°
Temperature: 74°
Collision: Error

Figure 4.5: HoneyBot user experiment website.

The real merit of the study, which is what the research participants were not told, is that the danger signs mark ”shortcuts” through the online maze and the navigation task cannot be completed in the 60 second time limit without cutting through at least two of them. The danger signs can be thought of as the honey or vulnerable resource on a real system/network tempting attackers to compromise it. The real maze contains walls where the danger signs are located and if they decided to take the ”shortcut”, the real robot would stop actuating through the real maze and simulate all further interactions. After cutting through a danger sign the online GUI robot continues through the maze ”normally”, but all data output to the sensor control panel from that point on is spoofed. After completing the experiment participants are given a survey, shown in Appendix C, and asked about the choices they made and what happened during the experiment.

4.2.1 Proof of Concept HoneyBot Sensor Model Development

Empirical observations were used to build the *Device Models* for the proof of concept HoneyBot sensors. Appendix D contains the Python script used for the sensor data collection process. For the temperature sensor, compass, laser distance sensor, and ultrasonic sonar the model development process was as follows:

1. The HoneyBot was placed at a viable maze location. For instance B from the labeled maze in Appendix B.
2. The Python script from Appendix D was executed and given the robots' coordinates in the physical maze. After that the program polls the robots' sensors for data values.
3. The script then creates an index in a CSV file with the given coordinates and adds the sensor values to the index.

This process is repeated several times at each of the 60 viable maze locations. A ”viable maze location” is defined as an allowable maze location for the robot to navigate to. Once these models were built the collision sensor device model was very simple. Since the robot was not allowed to perform commands that could actually cause it to crash, the only time the collision sensor needed to read ”True” was when the robot ”cut through” a danger sign. To do this the actual reading from the collision sensor (”False”) was always outputted to the user, unless they ”cut through” a danger sign. At that point the collision sensor outputted ”True” and the ultrasonic sonar/laser distance sensor outputted 0 for consistency. This was to really create the illusion that the robot hit an obstacle, but managed to keep going.

4.3 Experiment Results

The purpose of the research experiment was to evaluate the HoneyBot and determine how convincing the Sensor *Device Models* developed from real observations were. Of particular interest in the study were participants who ”cut through” danger signs to complete the maze quicker, because that action automatically triggered the *Input Verification Module* of the Robot Controller, which stopped the real robot from actuating and initiated the simulation.

4.3.1 Research Subject Demographics and Statistics

The research experiments took place on the Georgia Tech Atlanta Campus over the course of one week and was performed by 40 individuals from various academic/cultural backgrounds, physical locations across the US, and stages of life. The vast majority of subjects (95%) were young adults between the ages of 18 and 26. Figures 4.6, 4.7, and 4.8 give some quick statistics about the research subjects, including their regional location and cultural background.

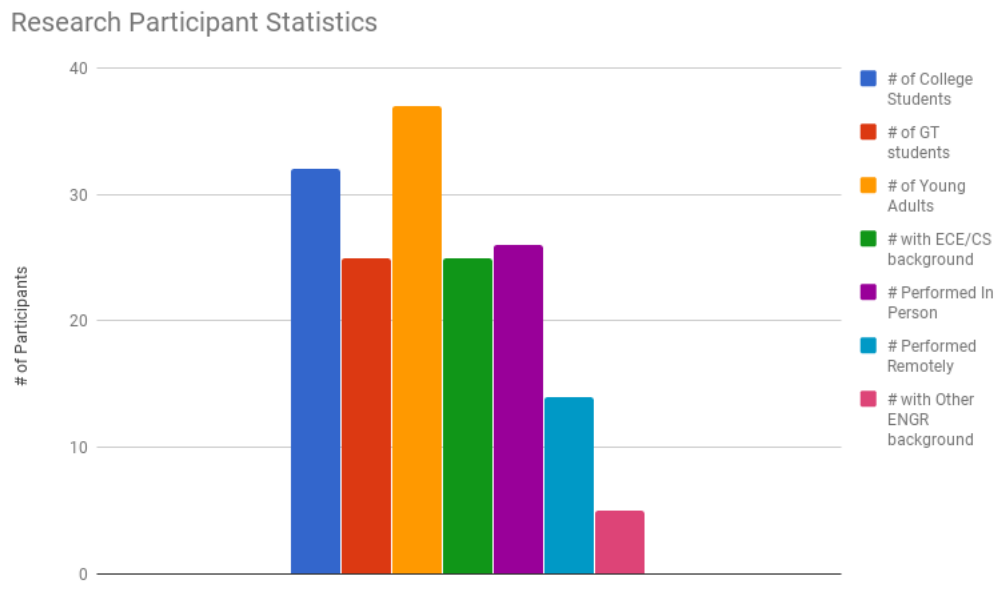


Figure 4.6: HoneyBot user experiment research participant statistics.

4.3.2 HoneyBot Experiment Findings

The HoneyBot experiment took approximately 20 minutes per participant and consisted of an initial instruction overview, the actual experiment completion, and a concluding survey. The Robot Experiment Survey, found in Appendix C was distributed through Qualtrics Online Survey Software, and consisted of, at most 12 questions. Certain questions were displayed/omitted based on

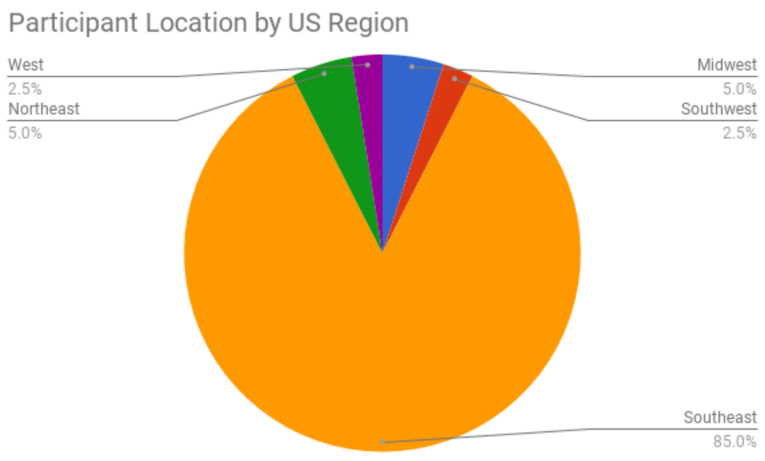


Figure 4.7: HoneyBot user experiment research participant locations by US region.

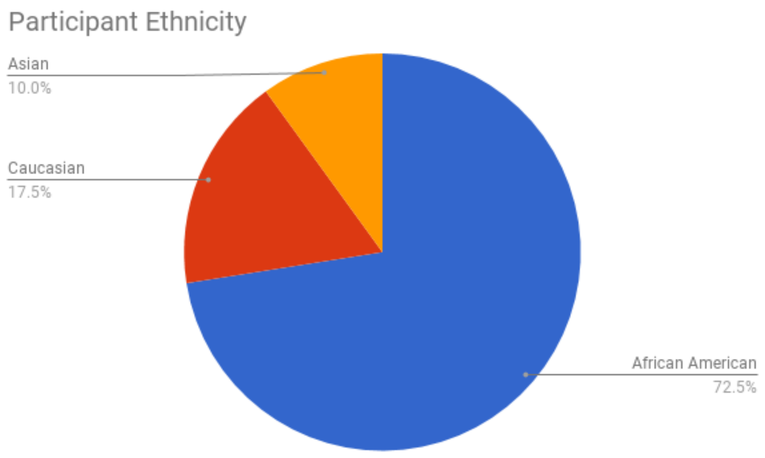


Figure 4.8: HoneyBot user experiment research participant ethnicity breakdown.

participant responses. For example, if Question 4, 'was the overall experiment completion process difficult?', was answered 'No' then Question 5, 'what made it difficult?', was not asked. Five questions on the other hand, were always asked. They served to provide baseline knowledge about the subjects' experience. The questions were:

1. Were you able to navigate from start to finish of the maze within the time limit?

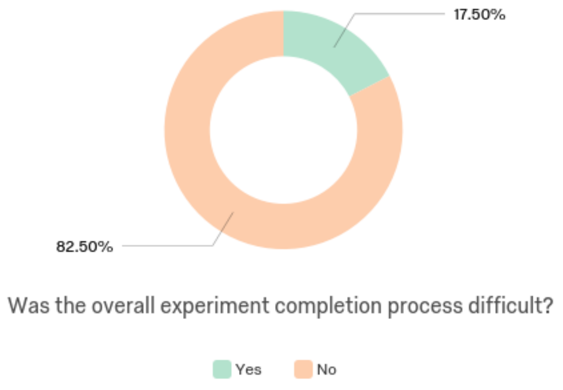
2. Map your navigated route by selecting the letters on the graphic below. If you did not finish the maze select to the nearest point you reached.

3. On a scale of 1-5, where 1 is very inaccurate and 5 is very accurate, how accurate did the sensor values displayed on the control panel seem throughout the experiment?

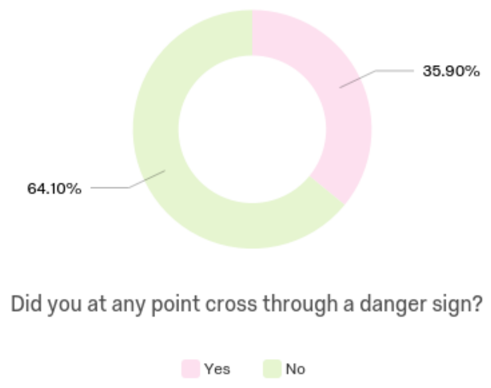
4. Was the overall experiment completion process difficult?

5. Did you at any point cross through a danger sign?

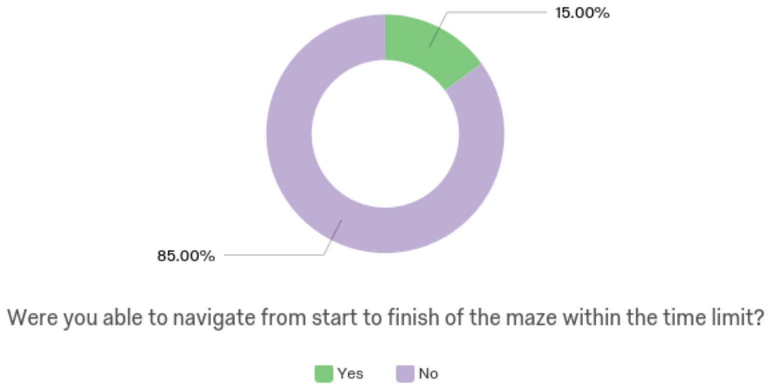
Figure 4.9 shows the user responses to survey questions 1, 4, and 6 from Appendix C. It can be gathered from the pie charts that the overall experiment completion process was not difficult, most people did not finish the maze in the allotted time, and a little over a third of the participants (14 people) ”cut through” at least one danger sign and were shown simulated sensor values.



(a)



(b)



(c)

Figure 4.9: Survey responses to (a) Question 4 (b) Question 6 (c) and Question 1 from Appendix C.

Question 2 came with the labeled maze shown in Appendix C, and according to the survey results, two of the three most traveled paths ”cut through” danger signs. 79% of subjects took the top three most navigated paths and the only other consistently navigated path (taken by 3 participants) also ”cut through” a danger sign. It is important to note that the routes depicted in Figure 4.10 indicate ”attempted” navigated routes, most subjects did not complete the maze, but indicated that was the route they intended to take.

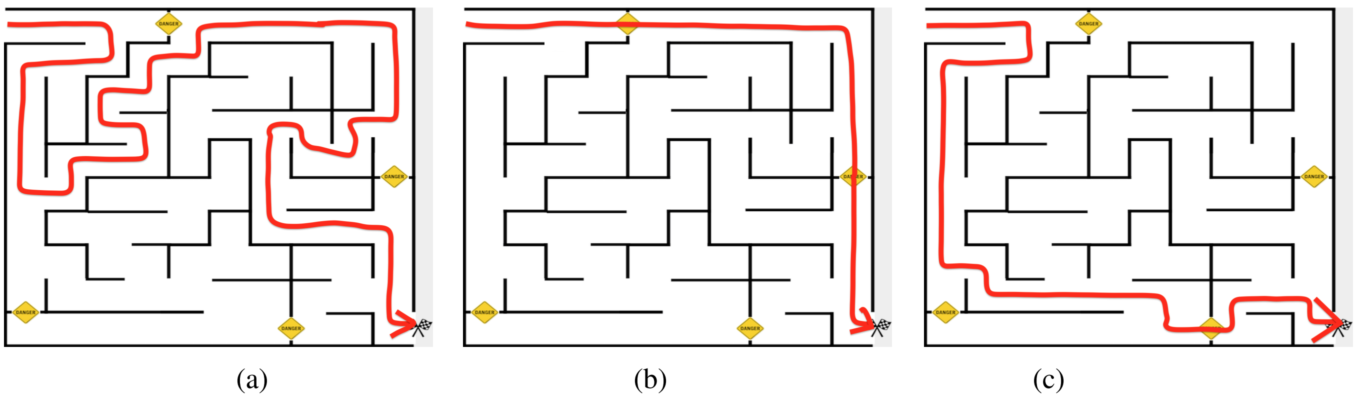


Figure 4.10: Top three most navigated routes by participants (a) 56% of subjects took this route (b) 22% of subjects took this route (c) and 1% of participants took this route.

In total, 14 participants cut through danger signs and triggered the HoneyBot ’simulation mode’. Table 4.1 shows that of all 40 participants surveyed, 70% of them rated the sensor accuracy during the whole experiment a 3 or 4 (mean of 3.58) out of 5. And of the 14 participants who cut through a danger sign and unknowingly experienced simulated sensor values, 71% rated the sensor accuracy a 4 or 5 (mean of 3.86) out of 5. This can be interpreted to mean research subjects did not notice a difference between the simulated sensor values and the real sensor values coming from the HoneyBot. From this it can be concluded that the proof of concept HoneyBot developed successfully fools ”deviant users” and the Sensor *Device Models* effectively mirror reality.

Table 4.1: Survey results to questions about robot sensor accuracy

Scale (1 is very inaccurate and 5 is very accurate)	How accurate did the sensor values displayed on the control panel seem throughout the experiment?	How accurate did the sensor values displayed on the control panel seem after you crossed through danger sign(s)?
1	1 (2.5%)	1 (7.14%)
2	4 (10%)	0 (0%)
3	13 (32.5%)	3 (21.43%)
4	15 (37.5%)	6 (42.86%)
5	7 (17.5%)	4 (28.57%)
Total	40 (100%)	14 (100%)

4.4 Limitations

The proof of concept HoneyBot implementation presented in this thesis serves as a show of feasibility for the concepts introduced. It is in no way a production ready implementation and has several limitations.

4.4.1 HoneyBot Website

The website used to remotely access and control the proof of concept was simplistic. It was built by manually constructing HTML, CSS, and JS pages as opposed to using a web framework like the Bootstrap toolkit. Bootstrap would have allowed for the use of a template/theme that would have erased the need to write HTML by hand. The look of the website was clunky and and a handful of users complained of a few non-intuitive features.

4.4.2 Robotic Platform and Sensors

The GoPiGo 3 was the chosen robotics system for the proof of concept HoneyBot and while it had many advantages, including its price point (approximately $200) and its plug and play sensor support, there were some down sides as well. Given its pricing the GoPiGo 3 was an economical robot targeting individuals wanting to try their hand at both programming and robotics. It is not a high end system for an avid roboticist, and reliability was an issue from time to time. This extends to the sensors as well, the manufacturer did not expect them to be used in critical infrastructure and thus they were prone to 'soft errors', such as randomly outputting erroneous data. A higher end robotic platform would be considered for more consistency and reliability.

4.4.3 Evaluation Caveats

The robot experiment evaluation, though it proved the proof of concept HoneyBot was convincing should be taken with a grain of salt. The small sample size of 40, is not enough to draw far-reaching conclusions. More user testing needs to be done to solidify the preliminary conclusions drawn. In addition to this, while there were some safeguarding (research proctors monitored the experiment task) against user falsifying the self reported survey results, there is always the possibility of fabrication when human subjects are involved and this must be considered.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

The need for security in the field of robotics will only grow in the coming years as robots are used for extensive tasks across various facets of life. Vulnerability and susceptibility to exploits in networked systems is unavoidable, and precautions must be taken to ensure that:

- The robotic system will be able to distinguish between safe and unsafe actions its commanded to perform
- The system uses this distinction to protect itself from physical harm
- There are reliable mechanisms means for system administrators to learn of compromise
- There are methods for gathering intelligence on system intruders

To help solve these problems, this thesis introduced the HoneyBot, the first honeypot specifically designed for robotic systems. The HoneyBot leverages HoneyPhy, techniques from traditional honeypots, and *Device Models* were built for common robotic sensors and queried at runtime to provide convincing system state updates and responses. By simulating unsafe actions and physically performing safe actions on the HoneyBot attackers are fooled into believing their exploits are successful, while logging all the communication to be used for attribution and threat model creation.

One underlying assumption behind the HoneyBot is that it lives on networks set up similar to the Georgia Tech Robotarium [37] in which users have no physical or visual access to the robotic system and send all commands remotely. Under this network configuration, the users only channel for keeping abreast of system changes are the real time system responses sent over the network.

The HoneyBot is a hybrid interaction honeypot specifically designed for robot systems and could potentially be the de facto standard for networked robot security as the prevalence of robots grow in society.

5.2 Future Work

The overarching goal of this research was to build a honeypot for robotic systems that could reasonably convince attackers that have remotely connected to a robotic system with malintent that their malicious payloads are successful, while in reality simulating data responses and preserving the real system. This was done through a proof of concept HoneyBot built on a GoPiGo 3 and was evaluated through a user study. While the preliminary results of the research study are promising, there is more work that can be done to improve the robustness of the ideas and implementations presented.

5.2.1 Exploring New Robotic Platforms

As described in the Section 4.4 the GoPiGo 3 robotic platform was sufficient for simple proof of concept research, but in order to extend the reaches of these ideas and frameworks, it should be replaced. A custom built robot that has more application to the manufacturing industry could be used. For instance, a robotic arm that must do a simple task such as pick up an object from one location and move it to another. This robot would normally perform the task on a loop, but also has the ability to be remotely accessed and controlled for maintenance purposes. An attacker could use this to throw off the entire assembly line process costing the organization hundreds of thousands dollars. In this example, the HoneyBot arm would detect the deviant commands and simulate the malicious actions all while logging and alerting system administrators of the compromise.

5.2.2 Rethinking HoneyBot Remote Access Mechanisms and Evaluation Redesign

With a change in the robotic platform would also come a necessary change in remote access techniques and new methods for evaluation. The proof of concept HoneyBot was accessible through a website which was functional, but had few usability issues. The new more industrial robotic platform may be better suited to be accessed via a command line tool such as SSH. If on the other hand a website is still found to be the best means, a more sophisticated design must be considered. The web server through which the proof of concept HoneyBot was accessed was run locally on the robot's Raspberry Pi, this introduced certain computational constraints and led to lag when many processes were running. To facilitate this, it would be imperative to run the web server for accessing the robotic system securely off-site and with enough resources to handle many web requests simultaneously.

Evaluation of the new platform would, of course, be heavily dependent of the main purpose of the new robot. Even still, the evaluation should not only involve human user testing, but general performance metrics as well. It would be important to note differences in response times of real vs simulated responses sent over the network, as well as any detectable footprint the HoneyBot software would leave on the system. In order to remain undetectable, processes should be hidden or run in an obfuscated manner very similar to a root-kit. Attackers/malicious parties who somehow gain full access to the system should not in theory be able to notice a difference between identical systems if one is running the HoneyBot software and the other is not.

5.2.3 Optimizing for Speed Over the Network

In the same vein of imperceptibility, random lag and general system unresponsiveness plagued the proof of concept HoneyBot from time to time. On a new robotic platform (or even for the current one) techniques must be employed to combat this as much as possible. It is impossible to do away with all lag or connectivity problems on the Internet, but with more computational resources and applying well known web toolkits, the occurrences can be reduced.

5.2.4 Software Generalization

The final proposed upgrade to the HoneyBot would be to make the code less platform dependent. The ideal situation would be to have a framework that is plug and play with as many robotic systems (ground robots, drones, humanoids) as possible. For this to be realized there are a few things that need to be done:

- A common HoneyBot API should be created for robots to subscribe to
- A minimum base set of sensors which all HoneyBot robotic systems are required to have must be defined
- A minimum set of computational resources which all HoneyBot robotic systems are required to have must be available on board the robotic system
- An automated process for managing sensor model generation must be established

Platform independence will be the most challenging of the proposed updates to the HoneyBot, but the software challenges can be overcome through the specifications defined above. Robotic systems that meet the criteria will be supported and those that do not will be ineligible.

Appendices

APPENDIX A

HONEYBOT IPHONE 5 COMPASS APP SOURCE CODE

```
import UIKit
import CoreLocation
import Alamofire

class DataViewController: UIViewController,
   CLLocationManagerDelegate {
    var locationManager: CLLocationManager = CLLocationManager()
    var startLocation: CLLocation!

    @IBOutlet weak var magHeading: UILabel!
    @IBOutlet weak var trueHeading: UILabel!
    @IBOutlet weak var latitude: UILabel!
    @IBOutlet weak var longitude: UILabel!
    @IBOutlet weak var arrowImage: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = UIColor.lightGray

        locationManager.desiredAccuracy = kCLLocationAccuracyBest
        locationManager.delegate = self
        locationManager.requestWhenInUseAuthorization()
        locationManager.startUpdatingLocation()
        locationManager.startUpdatingHeading()
        startLocation = nil

        // Create the image for the compass
        arrowImage.image = UIImage(named: "arrow.png")
        self.view.addSubview(arrowImage)
        arrowImage.center = self.view.center
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
    }

    func locationManagerShouldDisplayHeadingCalibration(_ manager
: CLLocationManager) -> Bool {
        return true
    }

    func locationManager(_ manager: CLLocationManager,
didUpdateHeading newHeading: CLHeading) {
        if (newHeading.headingAccuracy > 0) {
            magHeading.text = String(format: "%.3f", newHeading.
magneticHeading)
```

```
            trueHeading.text = String(format: "%.3f", newHeading.trueHeading)

            let heading = -1.0 * Double.pi * newHeading.magneticHeading / 180.0
            arrowImage.transform = CGAffineTransform(rotationAngle: CGFloat(heading))

            let parameters: Parameters = [
                "trueHeading": Int(newHeading.trueHeading),
                "magHeading": Int(newHeading.magneticHeading)
            ]

            Alamofire.request("http://robotexperiment.hopto.org:5555/", method: .post, parameters: parameters, encoding: JSONEncoding.default)
            .validate()
            .responseJSON { response in
                switch response.result {
                case .success:
                    print(response)
                case .failure(let error):
                    print(response)
                    print(error)
                }
            }
        }
```

```
    }

    func locationManager(_ manager: CLLocationManager,
  didUpdateLocations locations: [CLLocation]) {
        let latestLocation: CLLocation = locations[locations.
  count - 1]

        latitude.text = String(format: "%.3f", latestLocation.
  coordinate.latitude)
        longitude.text = String(format: "%.3f", latestLocation.
  coordinate.longitude)
    }
}
```

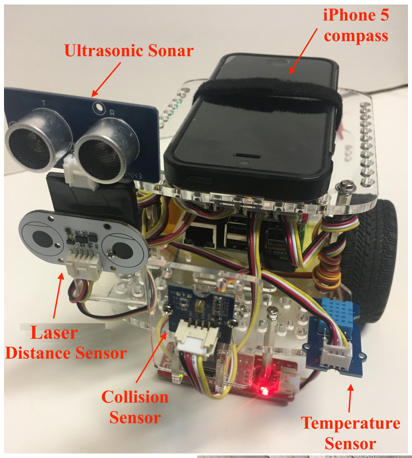
Listing A.1: RoboCompass App Source Code

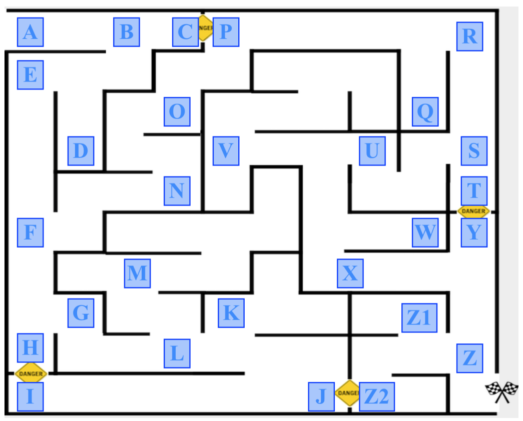
APPENDIX B
ROBOT EXPERIMENT INSTRUCTIONS

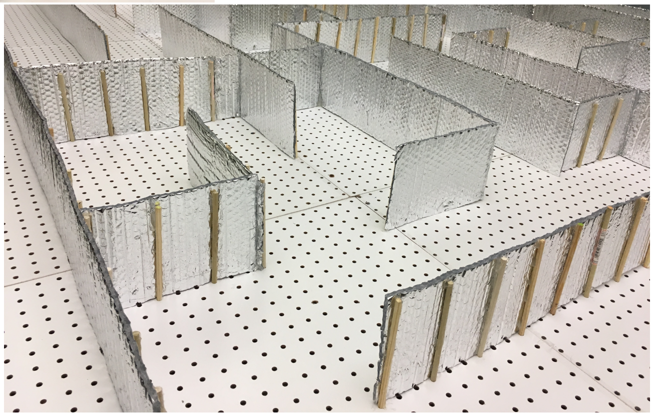
B.1 Overall Experimental Objective

We have built a robotic assessment course for testing the navigational abilities of a remotely accessible robot under various constraints. During this study participants will remotely access and guide a real robot through a real maze using an online virtual interface. After collecting sufficient experimental data, we will end the study and evaluate the results to determine the optimal constraint profile for the performance and efficiency of the robot.

B.2 Experiment Goals







This is Dex Jr. and his maze. Your task is to navigate as quickly as possible through the online virtual maze using the arrow keys (↑ ← ↓ →) on your keyboard to the finish flag. Dexs' sensor values will be available in an on-screen control panel (to the right of the maze) to help guide you and keep you aware of his physical status. **Make sure to pay close attention to the sensor control panel throughout the experiment, questions in the completion survey will ask you about it**. As you move GUI Dex through the online maze, the real Dex will move through the real maze in the same way.

B.3 Experiment Steps

1. Click the link to the experiment website. You will be directed to a web page asking you for a username and password.
2. **As soon as the correct credentials are entered the experiment will begin**. You will have 75 seconds to preview the maze and plan your navigation strategy. Make sure you consider **ALL POSSIBLE ROUTES** (even the risky ones) as you only have 60 seconds to complete the maze and the **robot moves VERY SLOW**. After you have strategized you must click the start button below the maze, if the timer runs out before you click start you will be unable to complete the experiment.
3. After clicking start you will have 60 seconds to complete the maze.
4. Once you've completed the experiment complete the survey at the link provided below.

B.4 Note

- **For navigating the robot through the maze, you only need to press an arrow key (↑ ← ↓ →) on your keyboard ONCE, when the robot is STATIONARY**. It will automatically make SEVERAL HOPS in the pressed direction to the next viable maze location and once it stops you can press the next arrow key. Pressing arrow keys more than once while the robot is moving will not make it move any faster. **If you press an arrow key (↑ ← ↓ →) while**

the robot is stationary and it does not move that means the robot is unable to move in that direction.

- Do not go backwards. This feature has been disabled because the task is timed and if you reverse you will not complete the maze.
- Do not refresh experiment website once youve logged in. If you do so, you will be locked out of the experiment.

APPENDIX C

ROBOT EXPERIMENT SURVEY

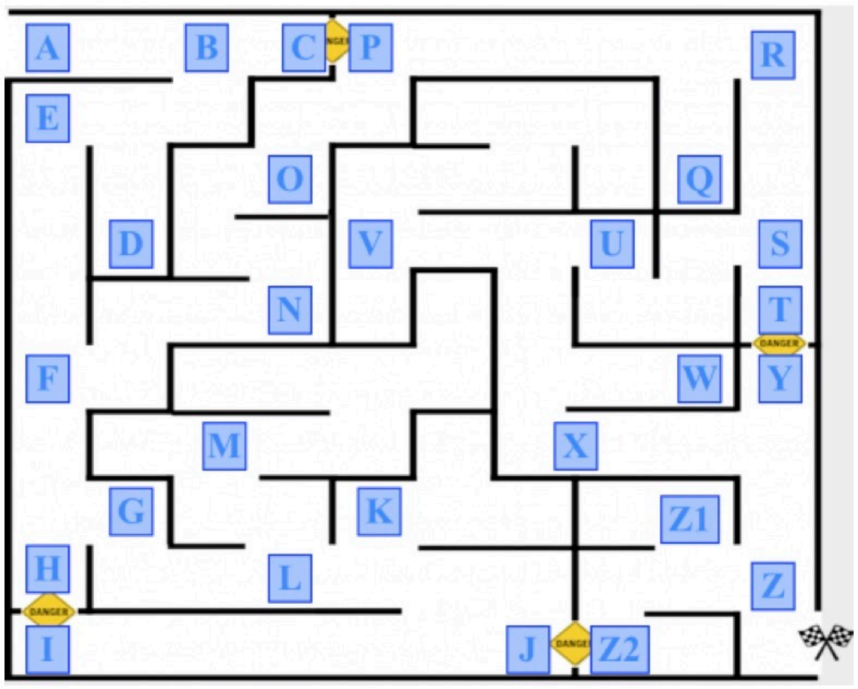
Q1

Were you able to navigate from start to finish of the maze within the time limit?

- ○ Yes
- ○ No

Q2

Map your navigated route by selecting the letters on the graphic below. If you did not finish the maze select to the nearest point you reached.



Q3

On a scale of 1-5, where 1 is very inaccurate and 5 is very accurate, how accurate did the sensor values displayed on the control panel seem throughout the experiment?

	1	2	3	4	5
Sensor Accuracy					

Q4

Was the overall experiment completion process difficult?

○ Yes

○ No

Q5

What made it difficult?

Q6

Did you at any point cross through a danger sign?

○ Yes

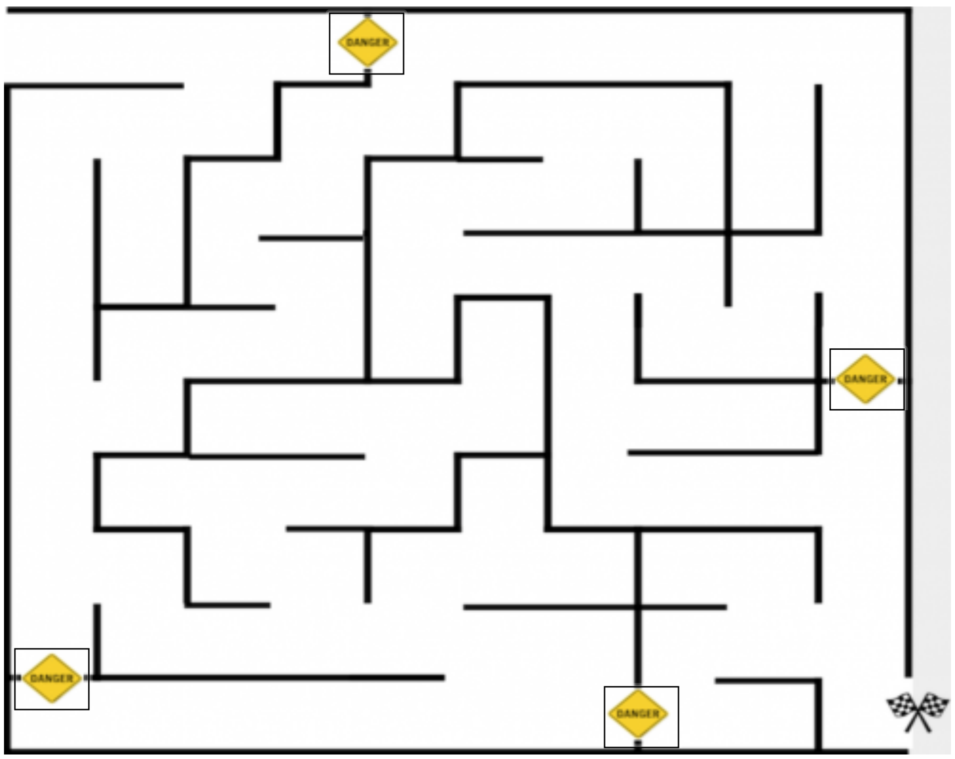
○ No

Q7

Why didn't you attempt to cross through a danger sign?

Q8

Select the danger sign(s) you crossed on the graphic below?



Q9

On a scale of 1-5, where 1 is very inaccurate and 5 is very accurate. How accurate did the sensor values displayed on the control panel seem after you crossed through danger sign(s)?

	1	2	3	4	5
Sensor Accuracy					

Q10

Why did you cross through the danger sign(s)?

Q11

Was there a difference between the GUI control panel values when you were navigating the robot after you crossed through the danger sign(s) versus before you crossed?

○ Yes

○ No

Q12

What difference(s) did you notice about the sensor values on the GUI control panel?

Powered by Qualtrics

APPENDIX D
SENSOR DEVICE MODEL CREATION CODE

```python
import sys
import pandas as pd
import  numpy as np
from pathlib import Path
import easygopigo3 as easy

gopigo = easy.EasyGoPiGo3()

deviceModelData = []
usPort = 'AD2'
collisionPort = 'AD1'
distancePort = 'I2C'
temperaturePort = 'SERIAL'


def get_user_input():
  try:
    inputX = input('Enter x coord: ')
    inputY = input('Enter y coord: ')
    inputCompass = input('Manually enter true heading: ')

    temperature_sensor = gopigo.init_dht_sensor(temperaturePort,
   0)
```

```python
    distance_sensor = gopigo.init_distance_sensor(distancePort)
    ultrasonic_sensor = gopigo.init_ultrasonic_sensor(usPort)
    reading = ultrasonic_sensor.read_inches()
    dist_reading = round(distance_sensor.read_inches(), 1)
    temp_reading = np.ceil(temperature_sensor.read_temperature() * 1.8 + 32)

    deviceModelData.append([(int(inputX), int(inputY)), reading, dist_reading, temp_reading, int(inputCompass)])
    print(deviceModelData[-1])
    return True
  except:
    print("Error. Ending")
    writeToCSV()
    return False

def writeToCSV():
  # Write device data to csv
  dataArray = np.array(deviceModelData)
  index = dataArray[:, 0]
  columns = ['Sonar', 'Distance', 'Temperature', 'Compass']

  dataVals = np.delete(dataArray, 0, axis=1)
  df = pd.DataFrame(dataVals, index=pd.MultiIndex.from_tuples(index), columns=columns)
  df.index.names = ['x', 'y']

  # Write to csv
  outCSV = Path("out.csv")
  if outCSV.is_file():
    df.to_csv(path_or_buf='out.csv', mode='a', header=False)
  else:
    df.to_csv(path_or_buf='out.csv', columns=columns, mode='a')

if __name__ == "__main__":
  try:
    out = True
    while out:
      out = get_user_input()
  except(KeyboardInterrupt, SystemExit):
    print("System interrupt")
    writeToCSV()
    sys.exit()
```

Listing D.1: Code used to collect sensor device model data and export to csv

REFERENCES

[1] Merriam-Webster. (2016). Robot, (visited on 11/01/2017).

[2] S. Webzell. (Feb. 2015). The rise of robotics, (visited on 11/01/2017).

[3] D. Kushner. (Feb. 2013). The rise of robotics, (visited on 11/01/2017).

[4] R. Langer, "To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve," The Langner Group, Tech. Rep., Nov. 2013, pp. 1–37.

[5] T. Bonaci, J. Yan, J. Herron, T. Kohno, and H. J. Chizeck, "Experimental analysis of denial-of-service attacks on teleoperated robotic systems," in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ser. ICCPS '15, Seattle, Washington: ACM, 2015, pp. 11–20, ISBN: 978-1-4503-3455-6.

[6] M. T. Review. (Apr. 2015). Security experts hack teleoperated surgical robot, (visited on 11/01/2017).

[7] M. Quigley, J. Faust, T. Foote, and J. Leibs, "Ros: An open-source robot operating system," in *Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, vol. 3, May 2009, pp. 1–5.

[8] H. H. King, K. Tadano, R. Donlin, D. Friedman, M. J. Lum, V. Asch, C. Wang, K. Kawashima, and B. Hannaford, "Preliminary protocol for interoperable telesurgery," in *Advanced Robotics, 2009. ICAR 2009. International Conference on*, IEEE, 2009, pp. 1–6.

[9] S. Litchfield, D. Formby, J. Rogers, S. Meliopoulos, and R. Beyah, "Rethinking the honeypot for cyber-physical systems," *IEEE Internet Computing*, vol. 20, no. 5, pp. 9–17, Sep. 2016.

[10] V. Pothamsetty and F. M. (Mar. 2004). Scada honeynet project: Building honeypots for industrial networks, (visited on 11/01/2017).

[11] L. Rist, J. Vestergaard, D. Haslinger, A. Pasquale, and J. Smith. (May 2013). Conpot, (visited on 11/01/2017).

[12] K. Wilhoit and S. Hilt, "The gaspot experiment: Unexamined perils in using gas-tank-monitoring systems," TrendLabs, Tech. Rep., 2015, pp. 1–22.

[13] Honeynet. (2017). The honeynet project, (visited on 11/01/2017).

[14] M. Dornseif, T. Holz, and C. N. Klein, “Nosebreak-attacking honeynets,” in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, IEEE, 2004, pp. 123–129.

[15] Honeynet. (2017). The honeynet project, (visited on 11/01/2017).

[16] P Defibaugh-Chavez, R Veeraghattam, M Kannappa, S Mukkamala, and A. Sung, “Network based detection of virtual environments and low interaction honeypots,” in *Proceedings of the 2006 IEEE SMC, Workshop on Information Assurance*, 2006, pp. 283–289.

[17] Desaster. (2017). Kippo - ssh honeypot, (visited on 11/01/2017).

[18] A. Morris. (Dec. 2014). Detecting kippo ssh honeypots, bypassing patches, and all that jazz, (visited on 11/01/2017).

[19] T. Holz and F. Raynal, “Detecting honeypots and other suspicious environments,” in *Information Assurance Workshop, 2005. IAW’05. Proceedings from the Sixth Annual IEEE SMC*, IEEE, 2005, pp. 29–36.

[20] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, IEEE, 2008, pp. 177–186.

[21] C. C. Zou and R. Cunningham, “Honeypot-aware advanced botnet construction and maintenance,” in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, IEEE, 2006, pp. 199–208.

[22] O. Hayatle, A. Youssef, and H. Otrok, “Dempster-shafer evidence combining for (anti)-honeypot technologies,” *Information Security Journal: A Global Perspective*, vol. 21, no. 6, pp. 306–316, 2012.

[23] M. L. Bringer, C. A. Chelmecki, and H. Fujinoki, “A survey: Recent advances and future trends in honeypot research,” *International Journal of Computer Network and Information Security*, vol. 4, no. 10, p. 63, 2012.

[24] M. Nawrocki, M. Wählisch, T. C. Schmidt, C. Keil, and J. Schönfelder, “A survey on honeypot software and data analysis,” *ArXiv preprint arXiv:1608.06249*, 2016.

[25] D. Industries. (2016). Gopigo, (visited on 11/01/2017).

[26] M. Zhang and X. Zhang, “Formally verifying navigation safety for ground robots,” in *Mechatronics and Automation (ICMA), 2016 IEEE International Conference on*, IEEE, 2016, pp. 1000–1005.

[27] A. M. Zanchettin, N. M. Ceriani, P. Rocco, H. Ding, and B. Matthias, “Safety in human-robot collaborative manufacturing environments: Metrics and control,” *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2, pp. 882–893, 2016.

[28] L. Wang, A. Ames, and M. Egerstedt, “Safety barrier certificates for heterogeneous multi-robot systems,” in *American Control Conference (ACC), 2016*, IEEE, 2016, pp. 5213–5218.

[29] N. Hacene and B. Mendil, “Toward safety navigation in cluttered dynamic environment: A robot neural-based hybrid autonomous navigation and obstacle avoidance with moving target tracking,” in *Control, Engineering & Information Technology (CEIT), 2015 3rd International Conference on*, IEEE, 2015, pp. 1–6.

[30] D. Industries. (2016). Grovepi, (visited on 11/01/2017).

[31] Matthew. (Dec. 2014). Linearizing the sharp ir ranger with an arduino, (visited on 01/01/2017).

[32] *Wide-angle distance measuring sensor*, GP2Y0A21YK, Rev. 3, Sharp, Jan. 2005.

[33] R. Goodrich. (Oct. 2013). Accelerometers: What they are & how they work, (visited on 11/20/2017).

[34] R. P. Foundation. (2012). Raspberry pi 3, (visited on 11/18/2017).

[35] T. T. Authors. (2009). Tornado, (visited on 11/18/2017).

[36] A. I. Automation. (2017). Robotic collision sensors, (visited on 11/18/2017).

[37] G. Tech. (Jan. 2017). The robotarium by georgia tech, (visited on 01/01/2017).