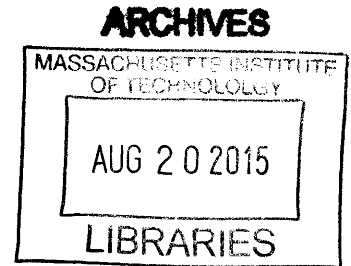
A Prototype System for Geo-Based, Cryptographically-Enforced Access Control for Miniature Drones' Video Feeds



by

Nathaniel A. Arce

S.B., Computer Science, MIT (2013)

Submitted to
the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author .... Signature redacted ....
Department of Electrical Engineering and Computer Science
September 2, 2014

Certified by... Signature redacted ....
Roger I. Khazan
Senior Staff, MIT Lincoln Laboratory
Thesis Supervisor

Certified by.. Signature redacted ....
Daniil Utin
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .... Signature redacted ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

A Prototype System for Geo-Based, Cryptographically-Enforced Access Control for Miniature Drones' Video Feeds

by Nathaniel A. Arce

Submitted to the Department of Electrical Engineering and Computer Science
on September 2, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we design and implement a robust proof-of-concept system for demonstrating the concept of usable, geo-based access control and agile, dynamic key management. The system utilizes a Parrot AR Drone 2.0 to stream an encrypted video feed to a number of Android-based tablets. The tablets are able to decrypt the video feed only if they are authorized to access it, based on the drone's location or a manual override by the drone's operator. As the individual tablets' access permissions change (either due to the drone's location changes or manual over-ride), the system enforces these permissions cryptographically through real-time, in-band rekeying of the authorized devices. This rekeying occurs virtually instantaneously, without any loss in the quality of service for the authorized participants.

The proof-of-concept system achieves two goals. First, it serves as a compelling demonstration of the Lincoln Open Cryptographic Key Management Architecture (LOCKMA) library. It illustrates how usable and seamless cryptographic protections can be straightforwardly utilized in an application, such as our geo-based drone prototype, using LOCKMA's intuitive interface for cryptography, key management, and access controls. Second, the proof-of-concept system lays the foundation for developing the geo-based access control concept further for drones and, possibly, other types of mobile data distribution systems. The software produced in this thesis project can also be used as a base for such future explorations.

This thesis document summarizes the project, the system architecture and its implementation, and lessons learned.

Thesis Supervisor: Roger I. Khazan
Title: Senior Staff, MIT Lincoln Laboratory

Thesis Supervisor: Daniil Utin
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

This thesis closes an important chapter of my life. I was an undergraduate at MIT as well, and in these last five years I feel that I have grown dramatically as a person. This isn't to say I'm really leaving; I plan to stay in the Boston area for many years to come, and most of my friends still live in the area as well. However, I am simultaneously happy and wistful to be ending my time working directly with the Institvte.

This thesis was funded by the MIT Lincoln Laboratory, Division 5, Group 58, through a research assistantship that I was fortunate enough to be offered.

My thesis advisor, Roger Khazan, has been a great motivating figure in this work; he has helped me in innumerable ways throughout.

Dan Utin, also of the MIT Lincoln Laboratory, is the primary developer of LOCKMA; he acted as a direct supervisor for the technical aspects of my thesis work.

Raymond Govotski, a contractor for the MIT Lincoln Laboratory, assisted me in learning to use LOCKMA over the course of the project, and was in charge of the ARM compilation of applications to be used in the Parrot AR Drone, itself.

Eric Grimson, Chancellor for Academic Advancement of MIT, previously Chancellor and once head of the Department of Electrical Engineering and Computer Science, has been my academic advisor since my time as an undergraduate, and was a major motivating factor in ever pursuing a Master of Engineering degree.

Contents

1 Introduction **13**

1.1 A Motivating Example 14

1.2 Contributions 15

1.3 Thesis Outline 16

2 Background Work **17**

2.1 Location-based Access Control 17

2.2 LOCKMA 17

2.3 Parrot AR Drone 2.0 20

2.3.1 Video Stream 20

2.3.2 Embedded Linux 21

2.3.3 Software Support 21

2.4 Technical Underpinnings 21

2.4.1 Node.js 22

2.4.2 OpenLayers 22

2.4.3 TileStache 22

2.4.4 Android SDK/NDK 23

3 High-Level Project Overview **25**

3.1 Top Level 25

3.1.1 Threat Model 28

3.2 Unmanned Aerial Vehicle 30

3.2.1 Key Management Process 30

3.2.2 Video Encryptor Process 31

3.3 Ground Control Station . . . 32
3.3.1 Node.js Server . . . 32
3.3.2 User Interface . . . 35
3.3.3 Key Management Center . . . 37
3.3.4 Video Decryptor . . . 37
3.4 Android Ground Terminals . . . 38
3.4.1 User Interface . . . 39
3.4.2 Backend Operations . . . 40
3.5 Design Conclusion . . . 41

4 Implementation Details **43**
4.1 Idiomatic Note . . . 44
4.2 Unmanned Aerial Vehicle . . . 44
4.2.1 UAV Key Management Process . . . 45
4.2.2 Video Encryptor . . . 48
4.3 Ground Control Station . . . 51
4.3.1 Node.js Application . . . 51
4.3.2 User Interface . . . 54
4.3.3 Key Management Control Center . . . 57
4.3.4 Video Decryptor Process . . . 60
4.4 Android Ground Terminal . . . 61
4.4.1 Android Application UI . . . 62
4.4.2 Video Stream Pipeline . . . 63
4.4.3 Key Management and Encryption . . . 65
4.5 Implementation Conclusion . . . 66

5 Conclusions **69**
5.1 Technical Results of LOCKMA . . . 69
5.2 Future Work . . . 72
5.2.1 Dynamically Adding New Ground Terminals . . . 73
5.2.2 LOCKMA for Key Management, IPsec for Encryption . . . 73

5.3 Learning Experience 74
5.3.1 New Technologies and Algorithms 74
5.3.2 Non-technical Lessons 77

A Utilized Non-Critical Components **79**
A.1 Proper Android Password Masking 79
A.2 iPhone-Style Auto/Manual Switch 81
A.3 Android CMake 81

List of Figures

1-1 Seamless and transparent security for drones. 14

2-1 A LOCKMA keywrap structure. 18

2-2 An overview of the LOCKMA API. 19

3-1 Notation used in Chapter 3. 26

3-2 Overall system interactions. 27

3-3 The Parrot AR Drone 2.0. 30

3-4 The UAV's process interactions. 31

3-5 The GCS's User Interface. 33

3-6 The GCS's process interactions. 34

3-7 The Manual Access Control Panel 35

3-8 The GTs' User Interface. 38

3-9 The GTs' thread interactions. 39

4-1 Notation used in Chapter 4. 43

4-2 The UAV KM Process's interactions. 45

4-3 The UAV Video Encryptor's interactions. 48

4-4 The GCS Node.js server's interactions. 52

4-5 The GCS User Interface's interactions. 54

4-6 The GCS KM Control Center's interactions. 57

4-7 The GCS Video Decryptor's interactions. 60

4-8 The GT's overall interactions. 62

4-9 The GT Video Stream Pipeline's interactions. 64

5-1 Scatterplot of LOCKMA decryption run-times. 71

Chapter 1

Introduction

In this thesis, we design and implement a robust proof-of-concept system for demonstrating the concept of usable, geo-based access control and agile, dynamic key management. The system is comprised of three types of wirelessly interconnected devices: a ground control station (GCS), an unmanned aerial vehicle (UAV), and several ground terminals (GTs). In our implementation, these devices are realized respectively as a Windows laptop, a Parrot AR Drone 2.0 UAV [1], and several Android-based Asus Eee Pad Transformer Prime tablets [2].

The GCS specifies, using an OpenLayers map, several geographical regions in which different ground terminals are authorized to access the UAV's video feed. The access permissions are enforced cryptographically: The GCS acts as a key management server, generating and securely distributing cryptographic keys to the UAV and the authorized GTs, in real-time, using the UAV's own wireless network. The UAV uses the key it receives from the GCS to encrypt its video feed. The GTs are able to decrypt the video feed only if they are currently authorized to access it. This depends either on the UAV's position on the map or, in the manual mode, by the GCS explicitly granting permission to a GT. All access control changes are implemented through real-time re-keying of the devices; such rekeying occurs virtually instantaneously, without any loss in the quality of service for the authorized participants.

Internally, our proof-of-concept system relies on the Lincoln Open Cryptographic Key Management Architecture (LOCKMA) library [3, 4]. LOCKMA is a software

component that provides a self-contained solution for easily integrating cryptographic protections and key management into applications. LOCKMA handles all of the necessary key management and cryptographic functions in a holistically architected and verified design, and provides a simple, intuitive interface to the application for invoking these functions. It supports agile, dynamic rekeying of individual devices and groups of devices, without requiring a centralized, enterprise-grade key server.

1.1 A Motivating Example

The use of drones for civilian purposes is a hot topic in the news today (see for example [5] and [6]). Video surveillance applications are one of the main uses, raising non-trivial privacy and security concerns [7].

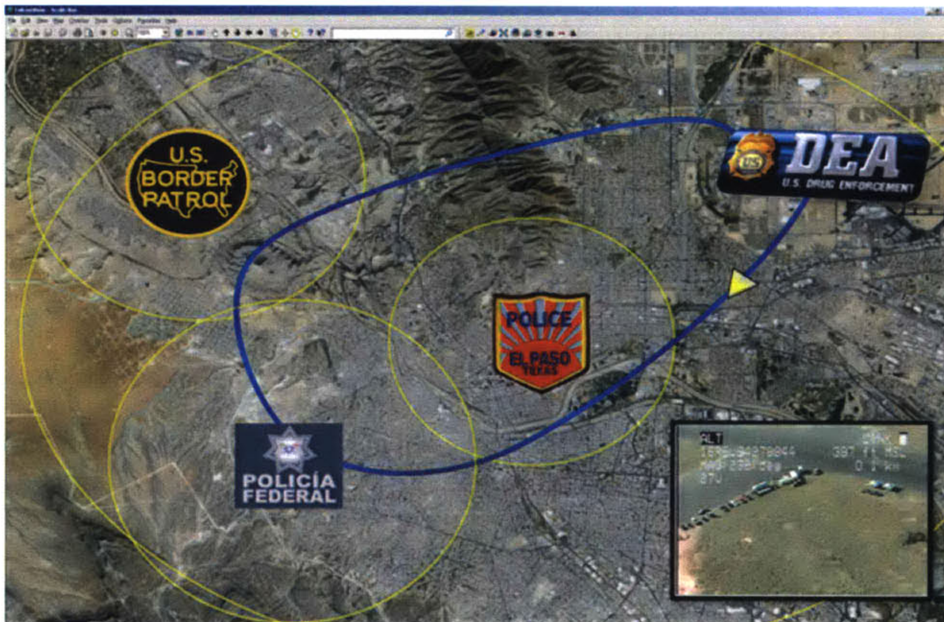


Figure 1-1: Seamless and transparent security for drones. The mission planner specifies the flight path and the geographical regions in which different participants are authorized to access the video feed.

Imagine a chase scene of the *Breaking Bad* nature[1], depicted in Figure 1-1. The

[1] *Breaking Bad* is a crime drama television show originally aired on AMC until 2013. The main character is, for most of the show, a producer of crystal meth, and as such is a natural target for the US DEA.

US Drug Enforcement Administration (DEA) has a UAV chasing a suspect (the UAV is depicted by the yellow triangle in Figure 1-1). The DEA is collaborating with various police departments, both domestically and across the border. It is, however, quite likely that up to a certain point, each police department does not want any others to get in its way or falsely assert a right to information. For protecting privacy and ensuring security of the mission, the DEA wants to only share the video feed on the need-to-know basis.

Figure 1-1 shows geographical regions where various police departments have jurisdiction and where the DEA would like to share the information with them. To make such access controls real, they have to be enforced through proper encryption. That is, the key that is used to encrypt the video feed at any given moment is made available only to a subset of all the ground terminals—those that are authorized to receive the video. For example, as the drone enters further into Mexico, the DEA would like assistance from Mexican Police; so the cryptographic key that is used by the drone to encrypt its video feed has to change and securely be made available to them. When the drone crosses back into the US, access permissions are modified to revoke Mexican Police's access and grant it to US Border Patrol.

1.2 Contributions

In this thesis, we designed and implemented a proof-of-concept system for the geo-based UAV video access control illustrated in Figure 1-1, using the OpenLayers library[8] as the user interface for rendering maps and specifying map overlays that correspond to access regions. This proof-of-concept system lays the foundation for developing the geo-based access control concept further for UAVs and, possibly, other types of mobile data distribution systems.

Specifically, the proof-of-concept implementation has undergone significant testing and resulted in a robust software system, which can be used as a foundation for future extensions.

Furthermore, the proof-of-concept system developed in this project serves as a

compelling demonstration of the LOCKMA library. It illustrates how usable and seamless cryptographic protections can be straightforwardly inserted in to an application, such as our geo-based UAV prototype, using LOCKMA's intuitive interface for cryptography, key management, and access controls.

1.3 Thesis Outline

This thesis document summarizes the project, the system architecture and its implementation, and lessons learned. Specifically, it consists of the following four chapters:

- Background Work - This chapter discusses the foundational technologies used in building the access control system. The chapter focuses most heavily on LOCKMA's functionality and design goals, but it also covers more widespread technologies in common use, and provides the technical underpinnings for development of the project.

- High-Level Project Overview - This chapter discusses the design of the project, including each component relevant to a technical demonstration. The design goals of the project are covered, along with the general execution plan of the project and how the foundational technologies fit together in the plan. It also discusses user interaction with the tool as a whole, delivering the idea of usable access control through the provided geographically-oriented map interface.

- Implementation Details - This chapter discusses the implementation of the project, going into the technical details of the work, both as it evolved and in its final form. Decisions for which specific technologies met the project's needs are discussed, along with technical issues the project ran into along the way.

- Conclusions - This chapter analyzes the work done, including a measurement of the overhead introduced by cryptographic operations on each end of the network. The chapter also discusses partially-completed and future work, and the potential merits of that work. Finally, it covers what the author has learned over the course of working on and completing this project.

Chapter 2

Background Work

Several technologies that were developed before my project began have proven critical to the goal of a highly usable access control system. LOCKMA and several other existing technologies came together to form a coherent demonstration that could, by design, be recreated rather precisely with the same technologies at one's disposal. In this chapter, we discuss these foundational technologies, going into detail where we feel it will be useful to the reader.

2.1 Location-based Access Control

The concept of location-based access control was introduced in a prior paper [9]. It was also a work developed by my advisors, Roger Khazan and Dan Utin. That work pre-dated LOCKMA and used a Department of Defense-specific mission planning tool, FalconView, as the user-interface. The prototype system ran on laptops, was not integrated into an actual UAV, and used a pre-recorded video file to simulate UAV video feed.

2.2 LOCKMA

The Lincoln Open Cryptographic Key Management Architecture, or LOCKMA [4], is the primary novel technology for this project. LOCKMA is a software component

designed to significantly simplify the task of adding cryptographic protections and underlying key management to software applications and embedded devices. LOCKMA utilizes NSA Suite B cryptographic protocols to achieve secure distribution of cryptographic keys and to encrypt and authenticate messages. Among these are AES block cipher modes of operation for encryption/authentication and Elliptic Curve Diffie Hellman for key agreement. The structure of a keywrap packet is found in Figure 2-1. Keys are distributed by encrypting to all specified machines using their key-agreement certificates or permissions granted by such certificates; thus, keys cannot be intercepted. Further, the key management operations are fast enough that there is no noticeable loss in immediately taking a system out of the loop by re-keying the other systems in use.

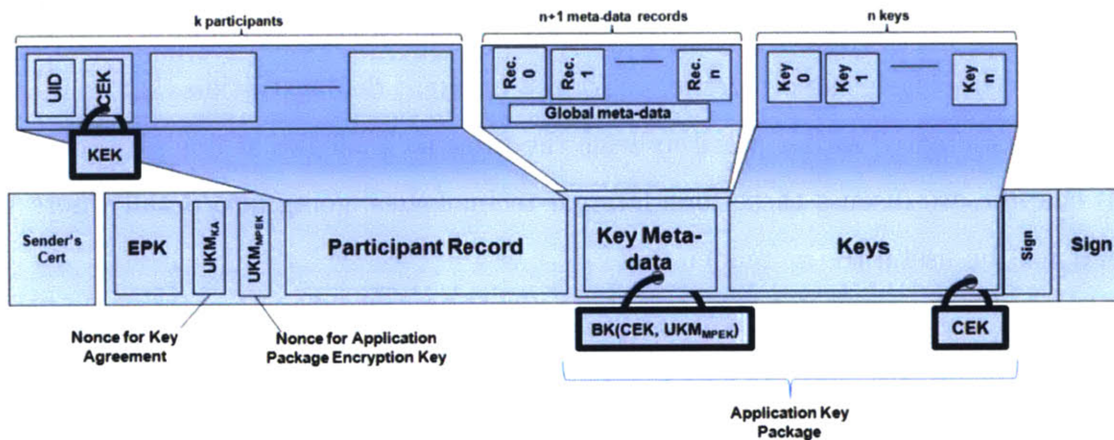


Figure 2-1: A LOCKMA keywrap structure. Keywraps are used to securely distribute cryptographic keys to several users at once. An Ephemeral Public Key and corresponding private key are generated; the private key is used to derive a key encryption key per shared user, in turn used to encrypt a singular content encryption key, which is finally used to encrypt the keys used for protecting and securing application data. In order to securely transmit these keys, Elliptic Curve Diffie-Hellman is used for key agreement to all target machines.

The LOCKMA API is specifically aimed to be friendly to users lacking advanced knowledge of cryptography. One of the core goals in its design and production is thus to make cryptographic security a more commonplace trait of daily computation and communication. Cryptography is becoming more widespread, but its adoption has

been slower than would be expected by masses of users concerned about their privacy. The primary issue there is usability. Fully open-source software has a tendency to become as labyrinthine as the developers can handle, but monetized software with simplistic user interfaces give virtually no granularity in what cryptography to use. Thus, LOCKMA finds an important niche in being usable by anyone familiar with, presently C programming, but at later points it will likely be even simpler to secure communications with LOCKMA, for example from the command line. A high-level overview of the API is detailed in Figure 2-2.

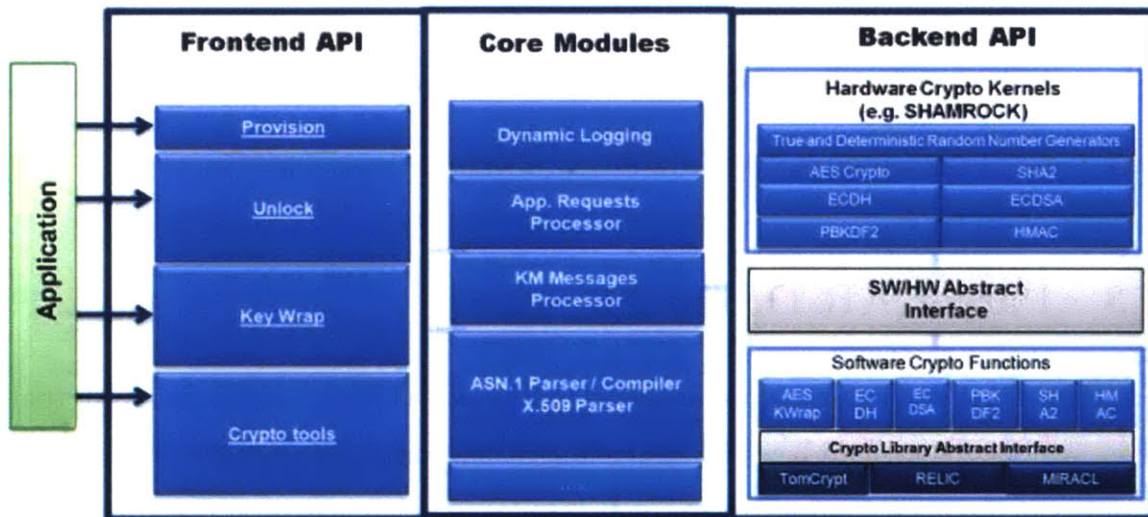


Figure 2-2: An overview of the LOCKMA API. Being an "Open Architecture" system, LOCKMA is designed to be able to be integrated into current and future systems, and can be easily updated to take advantage of newer cryptographic advances and methods. It does so without changing the simplicity allowed to the front-end developer.

Presently, LOCKMA's capabilities are detailed as follows:

- Identity Management
  - Generating and protecting long-term private keys
  - Exporting public keys and meta information as CSRs
  - Management of local user credentials that protect long term private keys
- Key Management

  - Request and authenticate remote device credentials
  - Generate and distribute key packages to groups of authorized entities

- Application access to common cryptographic functions
  - Digital Signatures
  - Key Agreement
  - Cryptographic Hashes
  - Key Derivation Functions
  - Application data protection: confidentiality, integrity, authenticity

- Support for software and hardware cryptographic backends

2.3 Parrot AR Drone 2.0

The Parrot AR Drone is a toy unmanned aerial vehicle (UAV), a quadcopter [1], that offers various digital interfaces for controlling it and, more importantly, features an output video stream from a camera built into its front. It has several features that made it desirable for a practical demonstration of LOCKMA. An image of the drone can be found in Figure 3-3, in the next chapter.

2.3.1 Video Stream

There is a camera embedded in the UAV that persistently takes a live video feed while the drone is turned on. There are also recording options built-in to the drone's software. A program can receive the video stream by simply connecting to the drone on a specific port; only one connection is allowed.

The video output is 720p, a high-definition stream, by default. While it is certainly easy for a demonstration to artificially generate more network traffic than that, an HD video stream is one of the highest sources of bandwidth that one could imagine requiring the delivery of in real-time. Therefore, this stream was ideal for testing

out the access control system's performance capabilities, encrypting and decrypting whole packets to verify that LOCKMA is usable in real-time with high-bandwidth traffic.

2.3.2 Embedded Linux

Parrot provides a Software Development Kit, but it is designed for software meant to interact with the drone, not to be placed on the drone. However, we were able to quickly determined that it runs an ARM processor with embedded Linux, meaning that the GNU C Compiler for ARM platforms is sufficient for the software we wanted to develop to place on the drone itself. If it had been running a customized kernel, a variety of issues could have arisen with our ability to interface software with the native system. For example, running Linux meant that all system calls were known to us. The file system was also structured in the expected way, such that configuring files to run as soon as the drone was powered on was fairly straightforward.

2.3.3 Software Support

In addition to the features of the UAV itself, there is a variety of software and support for interacting with the drone, some of which we made use of and will be discussed later in this chapter. This was made possible by the previously-mentioned Software Development Kit provided by Parrot.

2.4 Technical Underpinnings

An array of other technologies were utilized in this project, to varying extents. Some of these were more key to the technical content of the project than others, but all provided important utilities to the project as it was being developed.

2.4.1 Node.js

Node.js is a web server platform built on Google Chrome's JavaScript runtime [10]. It usefully supports asynchronous events and non-blocking input and output, and it is fast enough to not introduce appreciable delay in something like forwarding a high-bandwidth video stream.

There is an open-source Node.js project called node-dronestream that accepts the real-time video feed from the Parrot AR Drone and streams it to a browser window [11]. Using Chrome's graphical hardware acceleration and another Node.js package called Broadway.js, this module is able to stream the drone's video live without noticeable lag. By default, the node-dronestream application simply connects to the drone directly and broadcasts a web page to a specified port on the local machine that contains only playback of the video stream.

2.4.2 OpenLayers

OpenLayers is an open-source JavaScript software package that allows a dynamic map based on a supplied tile set to be easily included within a web page [8]. It offers useful features such as as custom graphical modifications, click-based events, and geometric intersection detection. Most notably of all is that it works in a fully offline environment, because its API can be stored locally. This is especially contrary to the Google Maps API, which requires being dynamically loaded through a web browser [12]. The only requirement for full offline compliance is to download or generate a tile set and serve it locally through a map tile server, so that maps do not have to be received through the internet. This tile server is covered in Section 2.4.3.

2.4.3 TileStache

TileStache is a Python-based server application that can serve map tiles based on rendered geographic data [13]. Through an open-use map tile host that offers free maps of various sizes from across the globe, we had access to nearly any map we could want, which could be easily converted into a format usable by TileStache. With such

a tile set available, TileStache can be easily configured to act as a tile server for any machine that knows how to connect to it, serving the specified tiles based on geographic coordinates.

TileStache accepts a variety of formats, though not the raw data supplied by our online source. As such, we made use of an additional free program called TileMill for generation of the tile sets that we then used with TileStache [14].

2.4.4 Android SDK/NDK

Android is a mobile operating system found on smartphones and other mobile devices [2]. It offers a software development kit and a native development kit for Java and C/C++ code, respectively [15]. These kits can be used to develop applications to run on the Android platform. Such applications can then be published to the Google Play Store for public consumption, or can be loaded locally onto any Android device with developer options enabled.

There is an open-source Android application called AR.Freeflight, developed by Parrot, that connects to all of the Parrot AR Drone's input and output sources [16]. It provides playback for the drone's video stream and allows the issuing of navigation commands. By the nature of the Android VM, all applications' entry points occur in Java, but through its heavy use of C code, Freeflight is able to stream the drone's video live without noticeable lag or other breaks.

Chapter 3

High-Level Project Overview

In this chapter, we discuss the design of the project as a whole. The implementation details can be found in the next chapter; in this one, we discuss the project from a broader perspective, making it clear how the technologies we used link together to achieve the goals of the project's core functionality. We also note certain steps in the evolution of the project over time.

Reading this chapter should be sufficient for the capability to implement a similarly-designed access control system. However, there are several technical details in the next chapter that are vital for streamlining the implementation process, especially for a system looking to utilize LOCKMA.

We will also provide diagrams to indicate interactions among the machines and processes involved. The notation used in this chapter is found in Figure 3-1. Due to the nature of discussion in this chapter, it differs somewhat from the notation found in the next chapter.

3.1 Top Level

In this section, we briefly discuss the overview of the demo's interactions as a whole. We do this in order to more smoothly transition into the design and functionalities of the individual components. The components' interactions with each other are the most important part of this project; therefore, it is important to avoid confusing the

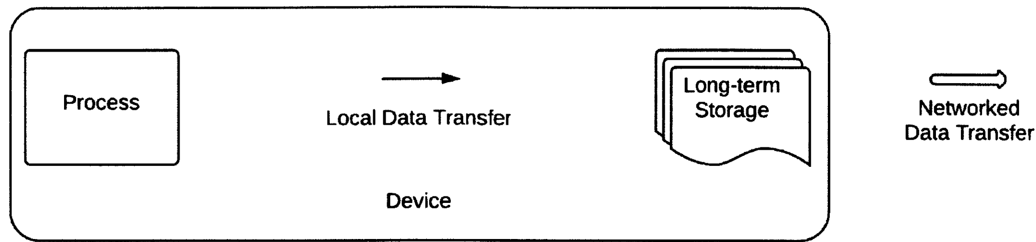


Figure 3-1: This diagram indicates the shape-based notation that will be used in all architectural diagrams for this chapter.

reader as would be caused by discussing a component's inputs and outputs without explaining the actions occurring on the other end.

Figure 3-2 summarizes the highest-level interactions among processes. There are three distinct components to this project: the Unmanned Aerial Vehicle (UAV), the Ground Control Station (GCS), and any number of Ground Terminals (GTs). In line with our story developed in Section 1.1, the UAV represents an entity controlled by the DEA, the GCS represents the machine utilized by the DEA official acting as the UAV's operator, and each GT represents a receiver of a local police department whose jurisdiction is indicated by the oft-mentioned map interface present on the GCS.

The demonstration as a whole was designed around delivering video to a subset of GTs specified by the GCS. Whenever the access list of GTs changes, all machines are sent a new packet containing an encrypted version of the new key list. If a GT is in the access list, they can decrypt the new content key using information stored by LOCKMA that comes from their private certificate. If a GT is not currently specified as having permission to access the video stream, it still receives data from the video stream, but it is unable to decrypt it.

The GCS acts as a central control unit for key management: it determines which machines should be allowed to decrypt the video stream, and it enforces this cryptographically by sending keywraps, packets that contain content keys that only specified machines can decrypt with their own long-term private keys. The UAV is the video source, in charge of all symmetric-key *en*cryption. It additionally acts as the delivery

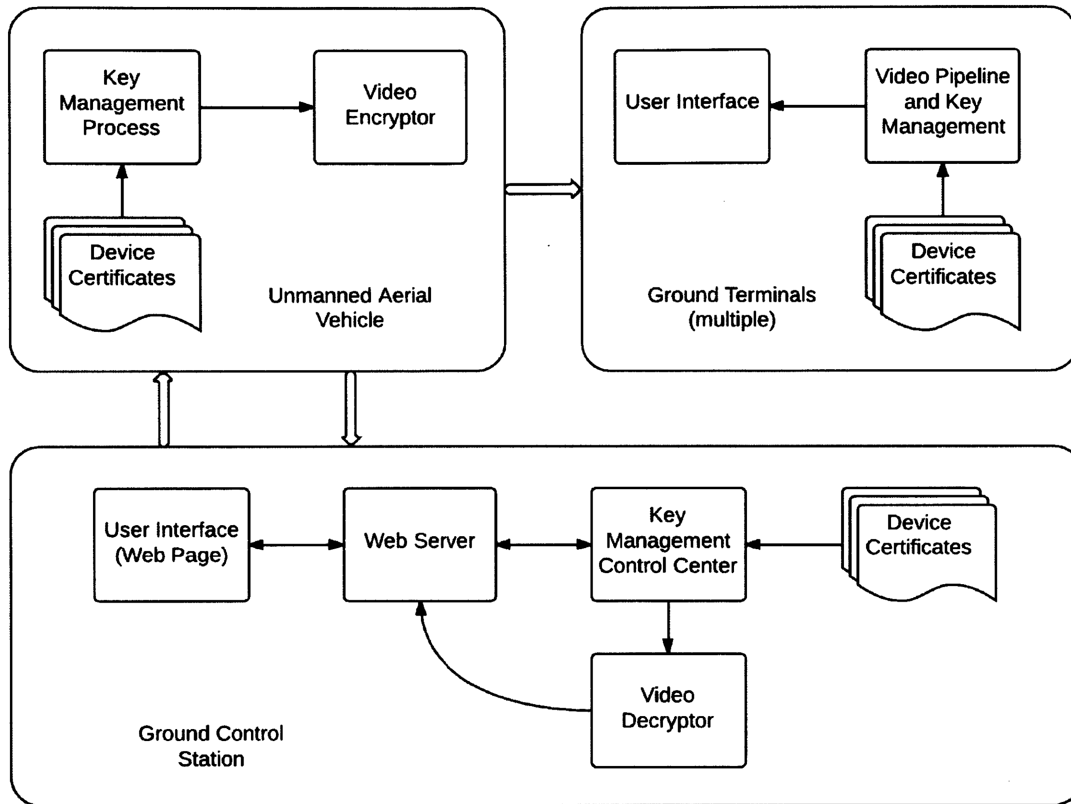


Figure 3-2: This diagram indicates all high-level interactions had among processes and devices in our demonstration. This diagram merely covers what interactions were present, rather than the details of the interactions. Each device will get its own design diagram with more details later on in the chapter.

mechanism for key management packets: the model here is that all machines that are capable of receiving video must be within receiving range of the UAV, but they do not necessarily have any other direct or indirect connection to the GCS. Thus, whenever the GCS constructs a new keywrap, it sends it exclusively to the UAV. The UAV then sends it to all listening parties, who can only decrypt it if they are on the GCS's access list. All of this takes place in a fraction of a second in the constrained-space environment in which the project demonstration was conducted. Thus, the switch to encrypting and decrypting using the new key found in the keywrap is able to happen very quickly.

3.1.1 Threat Model

As with any project related to computer security, the idea of a model of potential attackers on the system, or "adversaries", was considered and outlined before work on the project began. The precise threat model is not the most vital aspect of this project, due to the fact that our access control system can be generalized further than our demonstration went, and the project as a whole is not intended to cover any system-oriented attacks, which should be supplemented with other security measures. Thus, this project's threat model will be briefly covered here.

The single most important secure system at play for present and future variations of this demo, whose isolation is of the utmost importance, is the Public-Key Infrastructure (PKI) server, which distributes security certificates to all relevant parties so that key wraps can function as intended. However, during the actual running of this demo, the most important secure party is the GCS, which handles all of our key management in a centralized way. For the sake of the demo itself, the GCS may as well double as the PKI server, as it also keeps track of other parties' key agreement credentials to hasten key distribution. This secure system is the same that the administrative user operates, and it is the machine around which our access control system is centered. Therefore, this secure system, the GCS, being compromised is outside of the scope of this project; it is expected to be in a secure area and to not accept unnecessary traffic from external sources. If the GCS were to be compromised in the real world, the PKI server, a different machine that genuinely has no contact with the outside world, would distribute new certificates to a new GCS.

The UAV and other GTs, however, could be anywhere. By default, we assume to have knowledge and control over the UAV's location, but could be allowing it to wander as mobile surveillance. In the example described in Section 1.1, the UAV's movement would still be controlled by the GCS operator, but its movement would be dictated by necessity, not by a will to keep it away from potential attackers. Thus, its physical security must be ignored.

The video receivers other than the physically secured system, in other words,

the Ground Terminals, represent theoretical allies that we wish to share surveillance video with. We assume to have knowledge but not control over their location. As per Section 1.1, the two GTs we used represent the El Paso Police Department and the Mexican Federal Police.

Thus, at any given time, either the UAV or GTs could be physically compromised. We presume to have alternative communication set up among the operators of the GTs; if they fail to check in or are otherwise clearly compromised, we have the capability to permanently disable their access to the UAV's video stream. If the UAV itself is compromised, the mission must end but nothing relevant is lost; video does not enter the persistent storage of its source. In the current form of the project, the UAV will continue broadcasting using a now-compromised key until it is manually disabled or runs out of power. The loss in this case is the physical hardware and the device certificate; loss of hardware would certainly not be preventable by any degree computer security, and a long-term security certificate is considered a necessary storage element for LOCKMA's purposes. In some situations, loss of a security certificate could be considered a loss of money, as some certificate authorities can be expensive; in such cases, use of these certificates could be considered a detriment, or the certificate would need to be more heavily protected within the system. However, we assume for our purposes that certificates signed by our own central administrator are sufficient, and such certificates are therefore free.

We assume that attackers may attempt any degree of cryptographic threat against us, but currently NSA Suite B Cryptography is assumed to be secure. LOCKMA's cryptographic security is not within the scope of this project. Potential system exploits that can be incurred regardless of a system's unwillingness to accept SSH or Telnet traffic are also not in scope. If we were to become aware of such exploits, the previous actions for physical compromise of the machines would still apply; other systems can be responsible for the detection and prevention of such exploits.

In summary, the project's security goals are limited to the absolute security of the network traffic sent among the utilized devices.

3.2 Unmanned Aerial Vehicle

As discussed in Section 2.3, we utilized the Parrot AR Drone 2.0 in this project [1]. Our primary purpose for the drone was the HD video feed that it transmits to connecting machines. The UAV itself is pictured in Figure 3-3.

Figure 3-3: The Parrot AR Drone 2.0. This piece of hardware runs embedded Linux and has a single-core ARM processor operating at 1.0 GHz. Its capabilities and support made it an ideal choice for demonstrating LOCKMA's capabilities as a software component to be easily introduced into existing software.

This section discusses the design of the software we installed on the UAV. There are two distinct but communicating processes we added to the UAV, both of which we configured to launch on startup, whenever the UAV is turned on. A diagram detailing the high-level interactions within the UAV is found in Figure 3-4.

3.2.1 Key Management Process

On the UAV, the KM process is in charge of both receiving and sending KM packets. It communicates with three distinct entities: it receives from the GCS and sends to both the GTs and to its own Video Encryptor process. Near the beginning of its start-up phase but after establishing connections, it waits for a remote unlock

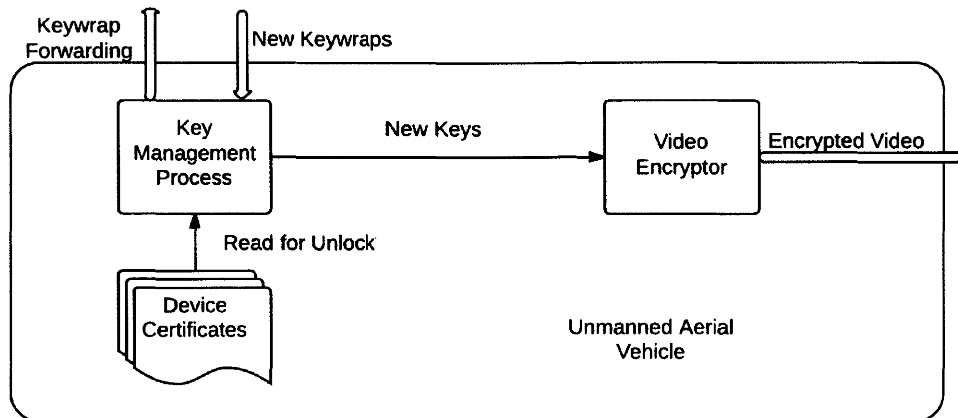


Figure 3-4: This diagram indicates all high-level interactions for processes running on the UAV. The UAV is responsible as a "go-between" to distribute keywraps from the GCS to the GTs, and is of course responsible for broadcasting its encrypted video stream.

command from the GCS's start-up phase; this remote unlock allows LOCKMA to be fully initialized on the UAV. After this, most of its time is spent merely waiting for input on its receiving socket.

When it receives such input, first it verifies its status as a key management message, then it passes off the packets wholesale to the GTs, so as to avoid sending a plaintext key or disturbing the message in a way that would disallow the GTs from decrypting it. Finally, it decrypts the key and sends the new content key to the Video Encryptor process locally. If it can't decrypt a key, it will continue using the old key, but this should not normally happen, as any key the GCS sends should include the UAV as a target recipient.

3.2.2 Video Encryptor Process

Due to the constraints of the UAV as it is commercially available, we could not programmatically redirect the video directly into LOCKMA, and instead intercepted it by connecting to the port locally. On Linux and most other operating systems, this is sufficient for our security goals, as it prevents the traffic from going through

the exposed network interfaces, thereby making it unable to be captured by packet-sniffing adversaries.

As such, the Video Encryptor's main function is to listen locally for the video stream, encrypt it, and send it off to all listening devices. It does this very frequently: the video streams at 30 frames per second. After each frame, it checks to see whether a key-containing message is also ready from the KM process. If so, it will accept and inject the new key into its own LOCKMA process, at which point it sets a short timer. This timer is to give the GTs enough time to decrypt and inject the relevant key as well: once the timer has finished, this process will begin encrypting with the new key. It will to encrypt with that key until it receives another one.

3.3 Ground Control Station

The GCS we used was a Dell laptop running Windows 7, but the technologies we used are multi-platform. Only the binaries we compiled from C could not be immediately reused on a machine running GNU/Linux, but these were easily configurable to be recompiled for other platforms.

The GCS is, unsurprisingly, where most of the human-computer interaction occurs, and where the most development was focused. A start-up script exists to launch all backend components, at which point the user interface can be accessed from a web browser, preferably Chrome for its graphics acceleration. Figure 3-5 shows the GCS's UI.

This section offers a breakdown of the software utilized on the GCS. There were four primary components, each of which we will discuss alongside any supporting programs. A diagram detailing the high-level interactions within the GCS is found in Figure 3-6.

3.3.1 Node.js Server

Building off of the node-dronestream project, the server's primary purpose is to serve the user interface to a local port. However, aside from simply serving the page to

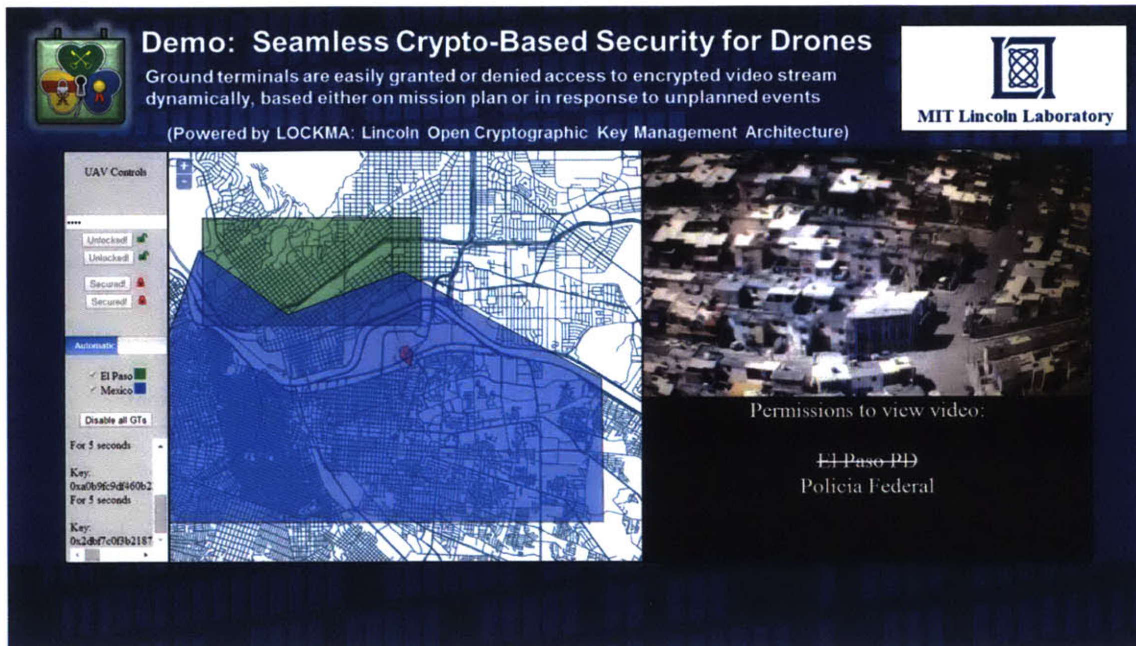


Figure 3-5: The Ground Control Station's User Interface. Commands are initiated with buttons located on the left panel. UAV virtual movement is controlled with the map interface. Video is streamed to the upper-right; the image on-camera is a photo of Juarez, a city in Mexico, for representational purposes. Current video access permissions are displayed in the lower-right area. Notice that the map is an analogue of the scenario presented in Figure 1-1.

a connecting web browser, this server needs four ongoing asynchronous connections to three different entities: two distinct connections are necessary to the web page it serves, one for streaming the video and one for acting on commands from the UI; then, one connection is necessary for each of the Video Decryptor and KM Center, the former for accepting the decrypted video stream and the latter once again for acting on commands from the UI.

Node.js is usefully event-based, so there is no need to wait on sockets. On start-up, connections are immediately established to the Decryptor and KM Center, and further action is impossible until the UI page is opened. When this occurs, both points of contact between the server and UI are established client-side (as the server doesn't otherwise know when the client is finished rendering), and video immediately begins streaming, getting passed directly through to the UI after a minor amount of header processing.

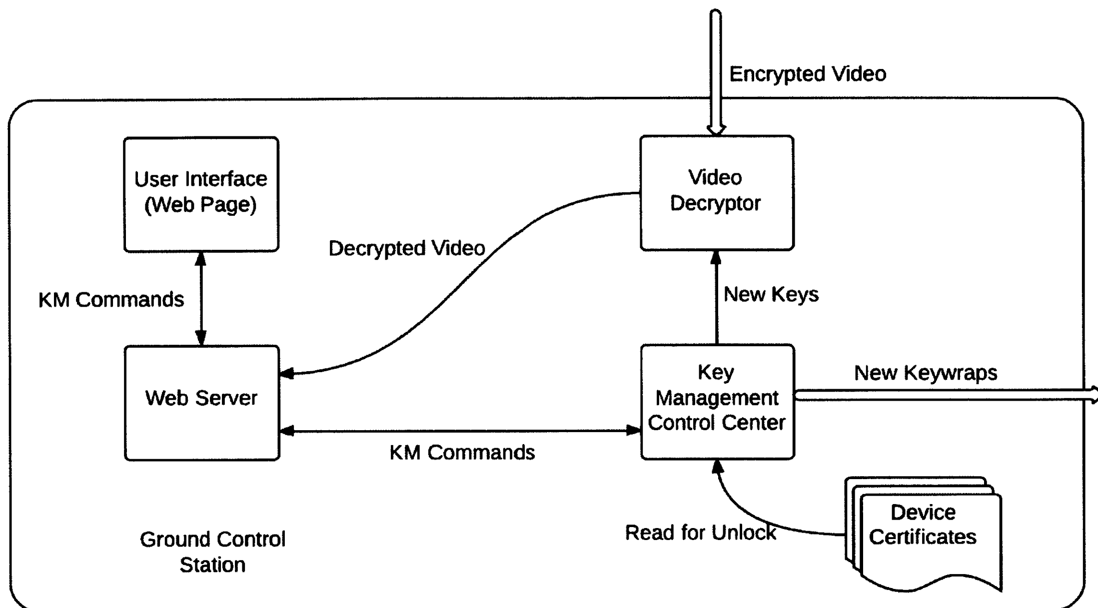


Figure 3-6: This diagram indicates all high-level interactions for processes running on the GCS. The GCS features a UI from which access control commands are issued, and acts on those by creating new keys for all devices to begin using for encryption and decryption. The UI is also responsible for playback of the video stream from the UAV.

At this point, the client using the UI is free to issue the commands available from the interface. When the Node.js server sees such a message, it determines based on a list of valid commands and its current state whether it needs to be passed to the KM Center or discarded. A discarded message based on an invalid command warrants an error message to be sent to the UI, as the UI should already lack the capability to send an anomalous command. If a message is instead discarded because state remembered from the GCS makes it no longer relevant, the UI is informed of this as well, so that it has the opportunity to catch up on the current state; this latter issue can come up if the UI web page was closed and re-started over the course of a demonstration.

The KM Center, which should only give information to the Node.js server after it has received commands, will send success or failure responses to the Node.js server that correspond to specific issued commands, sometimes with secondary information attached. All of this information gets transmitted to the UI to deal with, and in

the case of a success, such as in unlocking the device for use with LOCKMA, the aforementioned state variables are set within the Node.js server to indicate this. The server primarily functions as a go-between for the LOCKMA processes and the UI (without a good full-socket interface for an HTML page), so there is not a need for a heavy amount of processing involved.

3.3.2 User Interface

The UI for the GCS, and therefore for the Access Control system as a whole, is displayed in Figure 3-5. There are four key visual segments: the command interface, the map/controller interface, the video stream, and the video access list. The latter two are not directly interactive, but are important displays for the sake of the demonstration.

The command interface is used to start up LOCKMA on the GCS and the UAV. It's also used to manually override the automatic key signalling system provided by the map interface. The manual override system is displayed more closely in Figure 3-7. Lastly, the command interface displays the plaintext keys being utilized by the GCS as they get generated—again, we're trusting this machine to be secure.

Figure 3-7: The Ground Control Station UI's interactive panel for manually overriding access control permissions. Using this panel, an operator can make the system ignore current permissions that would be implied by the state of the map interface, by changing the switch from Automatic to Manual. From there, the checklist is enabled so that the operator may manually specify which GTs should have video decryption permissions. Clicking "Disable all GTs" automatically sets the switch to manual mode, along with un-setting all GT access permissions.

The map interface at the center of Figure 3-5 uses the OpenLayers API to provide a click interface for maneuvering the UAV in virtual space, so that a single click allows

the virtual UAV to travel in a realistic path towards the clicked location. In this way, we can direct the UAV into regions specified to be governed by specific GTs. The tiles are loaded from the local TileStache tile server. On the map, the regions of control are outlined and color-coded to clearly indicate which GT they belong to.

The video stream merely plays the video as it is taken by the UAV in real-time; there should never be a pause, because the GCS should always have all keys that the UAV uses for encryption. The video access list always displays the complete list of GTs, and it indicates which of them should currently have access to the video stream, according to the state of the UI itself, so one can examine whether this is reflected in reality on the GTs.

The first step must always be to unlock LOCKMA on the local device, by inputting the user password. This must go through the full process to the server then to the GCS and back; the full processing time is under a second. The user will be notified if the password was incorrect until a correct password is supplied, at which point the UAV may be given the signal to unlock. When success is reported for this operation as well, the map and command interfaces can each be used to determine how the video keys get redistributed.

Key distribution can be set here to automatic or manual mode. In automatic mode, the virtual location of the UAV determines which GTs have video access; if the UAV is overlapping one or more GTs' color-coded regions of control, those GTs should have a key to the video. In manual mode, a checklist is provided for each known GT to determine if it has access or not.

Whenever the access list changes, the UI merely sends the new list to the Node.js server, which forwards this information to the KM Center. Very soon after, the UI will receive a key associated with the new list, which it will print to the screen in hexadecimal.

In a real-space iteration of this project, the UAV's icon would not simply be controlled by point-and-click in the map interface; instead, its position within the interface would be detected by live GPS coordinates received from the UAV, which would be operated by an independent UI developed specifically for operating such a

device. Many such UIs already exist and behave effectively, so there would be no need to develop another one. With a sufficiently automated UAV, the possibility exists to control it using the map interface, but this could only be usable for basic surveillance, rather than our motivational DEA chase scene.

3.3.3 Key Management Center

The KM Center connects to three points: the Node.js server, the local Video Decryptor, and the UAV's KM process. These connections are the first to be set up on launch, after which the Center waits for input from the Node.js server. Nothing meaningful can be accomplished until LOCKMA gets unlocked with the local user password, so it waits for a successful unlock based on the server's supplied password. When this is complete and it has sent the success response to the server, it then waits for the word from the Node.js server to perform a remote unlock call on the UAV.

Finally, the KM Center enters its main loop, consisting of waiting for re-keying commands from the Node.js server. When it receives such a command along with an access list (that can be empty) of GTs to give access to the stream, it formulates a new key, sends that to the local Video Decryptor and to the Node.js server, and uses the signing credentials of the UAV and whichever GTs are on the access list to create a keywrap that can only be decrypted by the relevant parties. It then sends the new keywrap to the UAV for distribution to all listening GTs.

3.3.4 Video Decryptor

The Video Decryptor's direct connections have now all been covered: the Node.js server, the KM center, and the Video Encryptor on the UAV. Like the other C programs covered, the Decryptor sets up its connections first. When it starts receiving video packets from the UAV, it immediately begins decrypting them with the shared key, unless the UAV is still on from a previous demonstration and thus is using a different key, which is the only case for the GCS in which decryption should be able to fail. If and when a packet is successfully decrypted, it gets immediately shipped

off to the Node.js server.

Similar to the UAV's Video Encryptor, alongside each video frame, the Decryptor checks to see if a new key has come in from the KM Center, and if so, it injects the key into its LOCKMA instance so it can stay up-to-date with the UAV's encryption.

3.4 Android Ground Terminals

The core functionality of GTs is a strict subset of that of the GCS: they can stream video, and need LOCKMA user access, but cannot issue commands. In a previous iteration of this project, Ground Terminals were on Windows 7 PCs with their user interfaces being web pages in a similar style to the GCS. On Android, they function the same way but end up more visually distinct, with login access to LOCKMA required before the video stream even begins with the shared key. Both stages of the Android UI are pictured in Figure 3-8.

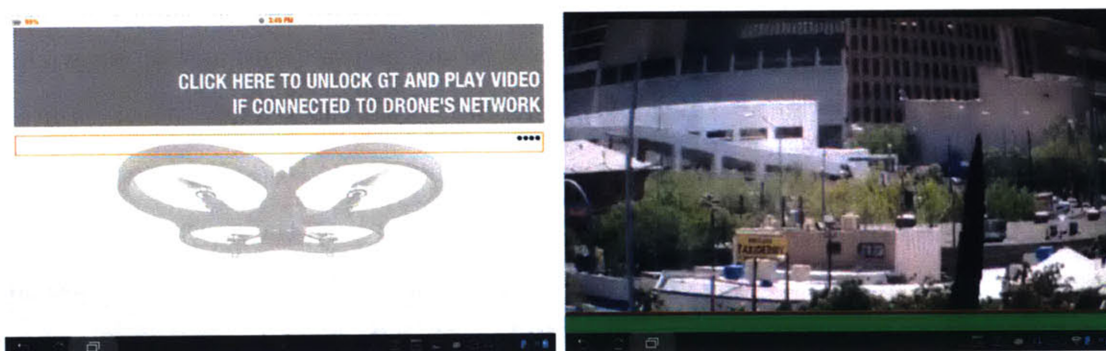


Figure 3-8: The Ground Terminals' User Interface. On the first screen, the user enters his/her LOCKMA password. On the second screen, the streaming video from the UAV is displayed when the terminal is authorized to receive it (i.e. has the correct video decryption key). The bars at the bottom of the GTs' video screens are color-coded to match the map in the GCS User Interface. The green bar indicates that this is the GT representing the El Paso Police Department, and the image on-camera is a photo of El Paso, again for representational purpose.

Android separates out a UI thread from anything involving network operations, and this dichotomy seems useful for separating our design overview, as well. A diagram detailing the high-level interactions within the GTs is found in Figure 3-9.

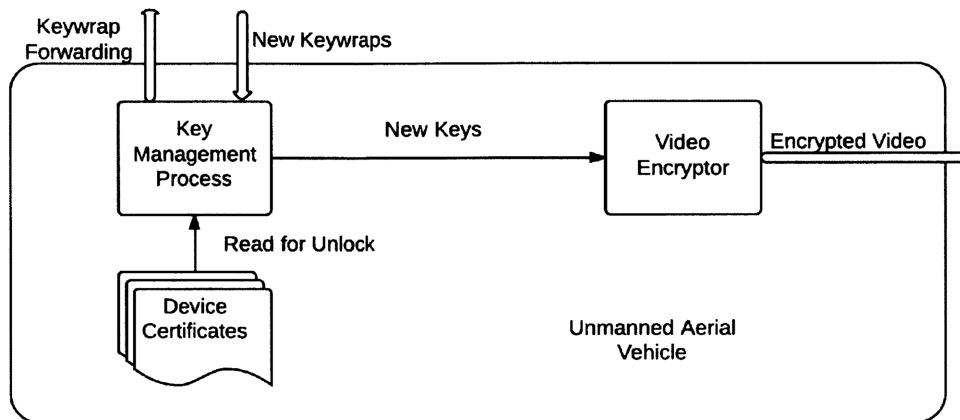


Figure 3-9: This diagram indicates all high-level interactions for threads found in the Ground Terminal application. The GTs act only as receivers, accepting new keywraps and featuring playback of the video stream whenever possible.

3.4.1 User Interface

As noted in Chapter 2, we built off of the Parrot AR.Freeflight open-source application to make this project. This seemed more beneficial as a demonstration of the LOCKMA component than making our own application from scratch because it best demonstrates the ease with which LOCKMA can be incorporated into existing applications, rather than requiring an application to be built around LOCKMA. As such, this UI is very familiar to those that have already used Freeflight; however, we have stripped out extraneous features and added a login functionality to the home screen, which is internally called the Dashboard.

When opening the application, the user is prompted to enter their LOCKMA user password in a box on the Dashboard screen. This application can run on smartphones, but our project was designed for tablets, so we decided to overwrite the default Android password box functionality: instead of displaying the last character entered into the box, all characters are hidden as they are typed, as expected of a password entry field on a PC.

When the correct password has been input, the user is taken to a screen only containing the video stream as it is being played. If a given GT is started early

enough in the project's timeline, the default shared key should allow streaming to begin immediately. While a GT's set of keys does not presently allow it to decrypt the stream, an "ACCESS DENIED" message is displayed, overlaying the video. This is to distinguish cryptographic enforcement from cases in which the connection to the video stream is lost or the UAV is shut down, which will merely cause the video playback to pause.

3.4.2 Backend Operations

We were required to utilize the Android NDK for C code, making use of a native thread for accepting video input from the UAV and also handling all decryption and Key Management messages. All of the new operations we added to the existing Freeflight code base were, for simplicity, added to the same function within the original Freeflight application. This function is the entry point where the video input was already being accepted on its appropriate socket.

In the end, these new operations were still able to be mostly converted wholesale from both the KM Process on the UAV and the Video Decryptor on the GCS. The existing application implements a multi-stage video pipeline, so the important aspect of this was getting video decrypted before the data got pushed to the socket stage's output, and making sure that no garbage output (so, no output at all) was pushed in the case that it couldn't be decrypted due to not holding the correct key.

In the case that the Android can suddenly no longer decrypt due to a key being incorrect, the process detects this and sends a notification to a Java thread set up specifically for this purpose. That thread then displays the "ACCESS DENIED" message on the video-playing UI, whose video should be paused due to lack of incoming decryptable video frames.

If the previous frame failed to decrypt but the current one succeeds, the UI is once again informed of this change so that the "ACCESS DENIED" message can be removed. We limit the frequency of these notifications by making it so that a notification will not be sent unless the decryption status changes—in other words, if the last frame was decrypted successfully, and this one was as well, there should not

be any new notification. However, even when sending the message once every frame during debugging, a performance drop was not observed.

3.5 Design Conclusion

In this chapter, we discussed, both generally and in depth, the design of each primary component of the project. The aesthetics of the available user interfaces were displayed, and the core functionality of each process running on each system had its internal functionality and external interfaces explained. As such, this chapter should have allowed a well-versed programmer to effectively recreate this project, given the same software resources.

Chapter 4

Implementation Details

In this chapter, we discuss the project on a more technical level. We discuss the requirements imposed by the technologies used, the steps taken during implementation, and the technical issues that had to be overcome along the way. We also provide code snippets from across all platforms of the project in order to aid understanding of some key areas.

Throughout the sections of this chapter, we utilize block diagrams to show the architecture of our processes. Figure 4-1 explains how to read the diagrams that appear.

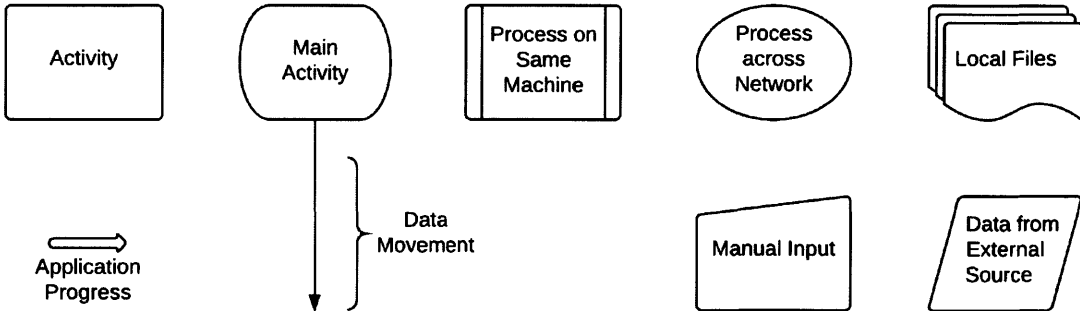


Figure 4-1: This diagram indicates the shape-based notation that will be used in all architectural diagrams for this chapter. Most of these are self-explanatory, but it is worth noting that once a main activity is entered, it is not ended until a demonstration is over. Applications that have a main activity exist for the purpose of that main activity, and all other activities are initialization steps.

4.1 Idiomatic Note

Before we begin our first implementation description, we'd like to mention one common idiosyncrasy among processes in this project: we used network sockets for all inter-process communication, rather than merely for communication to external devices. In some situations, this may not always be advised; however, for the common platforms, the fact that communication to localhost does not go through a publicly exposed network interface means that this is not a security issue unless a system has broken into one of our devices, which in our use case should already be considered a serious issue.

The primary reason that we chose to handle communication this way is for the useful symmetry provided by the select idiom. Namely, at least one process per machine wants to sit and wait in a loop until at least one source of input out of multiple has been detected. Generally, there will be two options, a key management message or a video frame; by waiting on both at once, we save performance and code compared to being required to check on each one separately.

4.2 Unmanned Aerial Vehicle

As stated in Section 2.3, the UAV has an ARM processor and runs embedded Linux. Thus, GCC for ARM devices is sufficient for compiling C programs to run on the device. We used the Sourcery Codebench for ARM/LINUX due to the array of libraries available roughly matching what is expected for x86 Linux.

As seen in the last chapter, there are two components running on the UAV, a Video Encryptor and a Key Management Process. Thanks to the expectations present on Linux, we were able to set these to start up as soon as the UAV is finished booting, by adding a line to /etc/init.d/rcS to run our startup script. The files were built by adding the source to CMake paths as part of the existing LOCKMA project, so that LOCKMA could get linked automatically.

4.2.1 UAV Key Management Process

The architecture for this process is found in Figure 4-2.

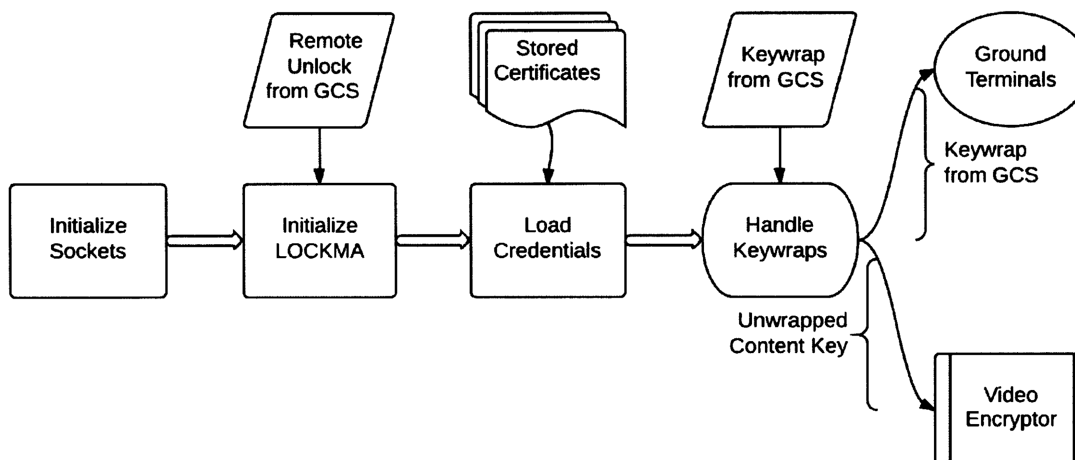


Figure 4-2: This diagram shows the trend of execution, the inputs, and the outputs of the Key Management process. Once LOCKMA is initialized, it repeatedly carries out its task of distributing and unwrapping new keys.

The KM Process starts up by initializing its three sockets: one TCP socket to connect to the Video Encryptor, a UDP socket for receiving KM Messages from the GCS, and another UDP socket for forwarding KM Messages to the GTs. LOCKMA initial set-up is handled as follows:

```
logger_init();
initialize_os_services();

memset((void *) &lockma, 0, sizeof(lockma_t));

lockma.pwd_entry_fn       = app_pwd_entry_fn;
lockma.rng_fn             = app_rng_fn;
lockma.read_config_fn     = read_config_fn;
lockma.write_config_fn    = write_config_fn;
lockma.app_free_fn        = free;
```

```
lockma.get_current_time_fn = get_current_time_fn;
lockma.app_pkg_notify_fn   = demo_app_pkg_notify_fn;
lockma.app_pkg_confirm_fn  = app_pkg_conf_fn;

result = lockma_initialize_instance( &lockma );
assert( result == LOCKMA_RESULT_OK );

result = lockma_process_config( &lockma );
if ( result != LOCKMA_RESULT_OK )
{
  do_provisioning( &lockma, SUBJ_INFO_COMMON_NAME,
                   sizeof( SUBJ_INFO_COMMON_NAME ) - 1 );
}

result = lockma_application_package_create( &lockma,
           KEY_PACKAGE_DEMO_META, sizeof(KEY_PACKAGE_DEMO_META),
           &app_package );
assert( result == LOCKMA_RESULT_OK );

memset((void *) crypto_channels, 0, sizeof(crypto_channels));
lockma.app_ctx = crypto_channels;
```

Some basic functionality of LOCKMA is initialized first. Then, LOCKMA's system and callback functions for standard functionality, most of which come with useful defaults from an existing test application, are set. Then, we verify that we have working configuration files, and give LOCKMA a blank slate for its current context.

After this, the process waits specifically for a remote unlock command from the GCS, as it is unable to unwrap any keys until it is unlocked. It simply waits on a select statement for this input. Eventually, the unlock will presumably succeed, or else the application never moves forward, and the process can load its device certificates for future decryption of keywrap packets. The reading of the certificates is done like so:

```
read_file( FILE_PATH_SIG_CERT, &cert_data, &cert_data_len );
lockma_add_device_certificate( &lockma, cert_data,
                               cert_data_len, &cert_type );
free( cert_data );

read_file( FILE_PATH_KAG_CERT, &cert_data, &cert_data_len );
lockma_add_device_certificate( &lockma, cert_data,
                               cert_data_len, &cert_type );
free( cert_data );
```

As such, LOCMA does all key extraction, and only needs to be passed file contents.

When this is complete, the only responsibility left is to wait on new keywraps from the GCS, in order to pass them on to the GTs. When a message is received from the GCS, first the UAV verifies that it is the expected type (else doing nothing), then performs a UDP broadcast of the packet so that the GTs may receive the keywraps as well. For unwrapping a key, the LOCKMA call is simply:

```
lockma_process_km_message( &lockma, data_ptr, data_len,
                           &km_message_response );
```

Then, in LOCKMA's app_pkg_notify_fn, which is called once a message has been processed, we extract the key to pass to the Video Encryptor:

```
lockma_key_mem_ptr key_ptr;
const uint8 *meta = NULL;
uint32 meta_len = 0;

lockma_application_package_get_global_meta( app_pkg, &meta,
                                            &meta_len );
lockma_application_package_get_key( app_pkg, 0,
                                    &meta, &meta_len, &key_ptr );
```

Thus, at this point key_ptr will hold the key that we can send through the local TCP socket to the Video Encryptor.

4.2.2 Video Encryptor

The architecture for this process is found in Figure 4-3.

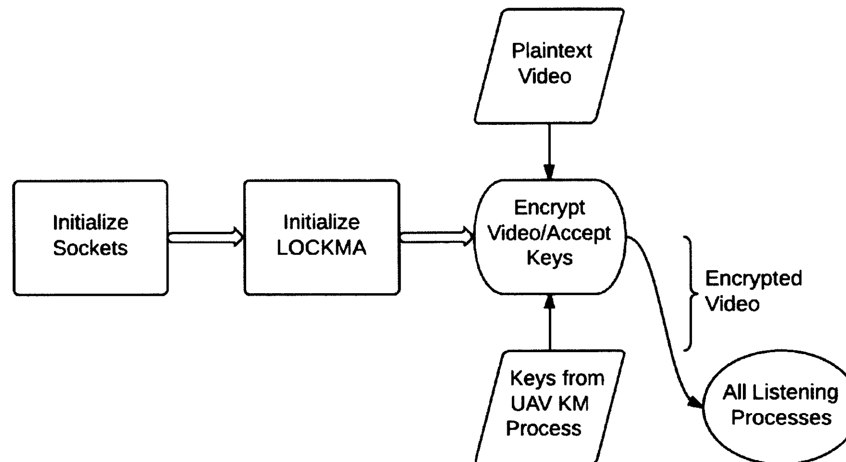


Figure 4-3: This diagram shows the trend of execution, the inputs, and the outputs of the Video Encryptor process. It occasionally has to accept a new key from the KM Process, but it is constantly encrypting video frames.

The Encryptor also starts up by initializing its three sockets: one TCP socket to monopolize the UAV's video stream output, one TCP socket to connect to the KM Process locally, and one UDP socket to output encrypted video. Because the Encryptor does not need to deal directly with KM Messages (only raw keys, from the KM Process), LOCKMA startup is simplified from the full startup seen previously, which included reading device certificates.

Once LOCKMA has been initialized, the terminal loop is immediately entered, in the described way characteristic of this project. The two input TCP sockets waited on with a select statement, as shown in the following code:

```
while (1)
{
  FD_ZERO( &read_fds );
  FD_SET ( video_fd, &read_fds );
  FD_SET ( km_fd, &read_fds );
```

```
  fd_count = select( highest_fd, &read_fds, NULL, NULL,
                     NULL );
  if ( fd_count < 1 ) {
    continue;
  }
  if ( FD_ISSET( video_fd, &read_fds ) )
  {
    // Receive, encrypt, and broadcast the video stream
    // from the program.elf process.
    broadcast_video_stream( video_fd, broadcast_fd );
  }
  if ( FD_ISSET( km_fd, &read_fds ) )
  {
    // Inject key request or encryption/decryption request.
    service_km_request( km_fd );
  }
}
```

Most variables and functions should be self-explanatory, but the others will be described.

The highest_fd variable is initialized before the loop to be the higher value between video_fd and km_fd, plus one. The broadcast_video_stream function accepts frames from video_fd, encrypts them using the current key index with LOCKMA, and sends the encrypted frames over the UDP socket broadcast_fd to listening devices.

Encrypting with LOCKMA is handled simply as follows:

```
LOCKMA_RESULT   result;
uint8           icv[LOCKMA_AES_GCM_AUTH_TAG_SIZE];
uint8           auth_data[APP_AUTH_DATA_LEN];
uint32          ciphertext_len;
uint8          *payload_ptr = *packet_ptr;
```

```
static uint32    sequence_nbr = 1;


lockma_encrypt_app_data_aes_gcm(
    aes_core.channels[active_channel_nbr].key, // key
    (const uint8 *) &sequence_nbr,   // iv address
    sizeof(uint32),                  // iv length
    plaintext_ptr,                   // plaintext address
    plaintext_len,                   // plaintext length
    (const uint8 *) auth_data,       // auth data address
    sizeof(auth_data),               // auth data length
    &payload_ptr,                    // ciphertext address
    &ciphertext_len,                 // ciphertext length
    icv );                           // Integrity Check Value

sequence_nbr++;
```

Buffers need to be allocated where appropriate, but the only user-supplied data that changes per run are the pointer to the unencrypted video frame (plaintext_ptr) and the associated length of the frame. As noted in the code, we are utilizing the AES GCM (Galois-Counter Mode) block cipher mode of operation, though LOCKMA supports many other symmetric-key algorithms as well.

The service_km_request function accepts a key from the TCP port connected to the KM Process. LOCKMA supports multiple keys in memory simultaneously so that the transition between two keys can be smooth and not require any dropped information, so this key is sent alongside the relevant channel number that the key should be used for. The function injects the key by simply copying its raw bytes into the corresponding channel slot, as follows:

```
recv_len = recv(km_fd, km_msg, sizeof(km_t), 0);
memcpy((void *) &aes_core.channels[km_msg.channel_nbr].key,
```

```
(void *) km_msg.key, SYMMETRIC_KEY_SIZE);
```

After this is done, a minor delay is set for setting active_channel_nbr to equal channel.channel_nbr, so encryption may begin using this key, as shown in the broadcast_video_stream snippet.

4.3 Ground Control Station

The GCS is a Windows 7 PC running an x86 processor. The C programs written for it were compiled with Microsoft Visual Studio 2010. Unlike the Ground Terminals, which better demonstrated direct integration of LOCKMA into an existing project, the GCS's C programs *were* built around LOCKMA: in general, the concept of a GCS in our access control use case primarily exists for the purpose of key management, so the GCS can act as its own standalone hub that does not truly require the ability to listen to incoming traffic. However, it was beneficial for us to construct a Video Decryptor for the GCS for demonstrative purposes, as the GCS is the only party that should always be able to decrypt the video stream.

This machine gave an opportunity to show off LOCKMA-built applications feeding into existing projects, as an alternative to integrating into applications directly as is the case with the GTs: all of our modifications to the Node.js application were for the sake of developing an accessible interface for issuing LOCKMA commands, and were not relevant to actual video decryption.

4.3.1 Node.js Application

The architecture for this process is found in Figure 4-4.

As is usual with Node.js applications, the first thing we do is load up the packages the server relies on. In this case, we used these packages: http, standard for serving to the web; dronestream, the primary package upon which our UI was based; buffy, as a useful parser for raw byte streams; net, useful for direct socket interfaces to other programs; and ws, a fast WebSocket package for dynamic communication between the

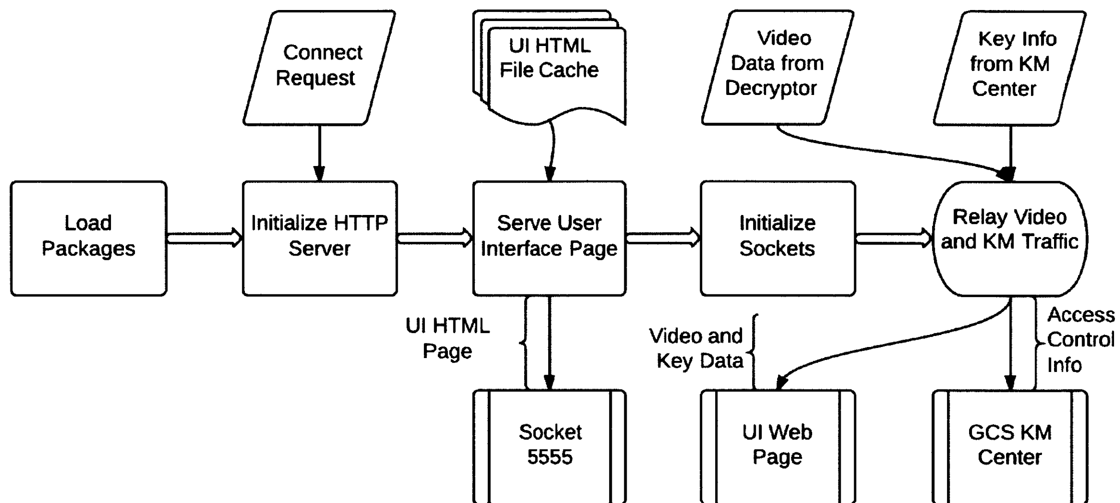


Figure 4-4: This diagram shows the trend of execution, the inputs, and the outputs of the Node.js server process. After loading the web page user interface, its primary purpose is merely as an information relay.

server and its hosted web page. We then set up the actual HTTP server, which simply pipes all files requested by a web browser so the web page can be reconstructed. Any requested file not present on the server is simply ignored.

We then set the dronestream package object to listen to our Video Decryptor (rather than the drone itself, which is its default setting), at which point the existing project handles all future communication with the Video Decryptor, which should only ever be sending properly-decrypted video. Finally, we make a new WebSocket Server (WSS) to /io so that web pages have a convenient interface for contacting the server. All connections to the Key Management Control Center are then also handled within the WSS, because without a web page to talk to, the Node server ferrying messages to and from the Control Center is not really useful to our demo. On an initial connection to the web page, the WSS begins reacting to both connections from the Control Center and the web page as follows:

```
ws.on('message', function(data, flags) {
  ws_opened = true; // The web page has finished loading
  values = JSON.parse(data);
```

```
  switch (values[0]) { // Each sends an appropriate message
                       // to the KM Center
    case 0: // Ground Control Station Unlock
            // comes with password
      ...
    case 1: // UAV unlock — no extra data
      ...
    case 9: // Re-key message.
      ...
  }
});

gcs.socket
  .on('data', function(buffer) {
    parser.write(buffer);
    which = parser.ascii(4);
    response = 0;
    // these each update the response variable to indicate
    // the Control Center response to a previous command
    if (which == "GCSU") { // GCS Unlock
      ...
    }
    else if (which == "UAVU") { // UAV Unlock
      ...
    }
    else if (which == "RKEY") { // Re-Key
      ...
    }
    //sends the response to the web page
    returnCall(which, response);
```

```
});
```

Simply put, these are convenient shorthand messages between the web page and the KM Control Center. Certain formats are easier for JSON and certain are easier for basic C string processing, and the Node server handles the transition step.

4.3.2 User Interface

The architecture for this interface is found in Figure 4-5.

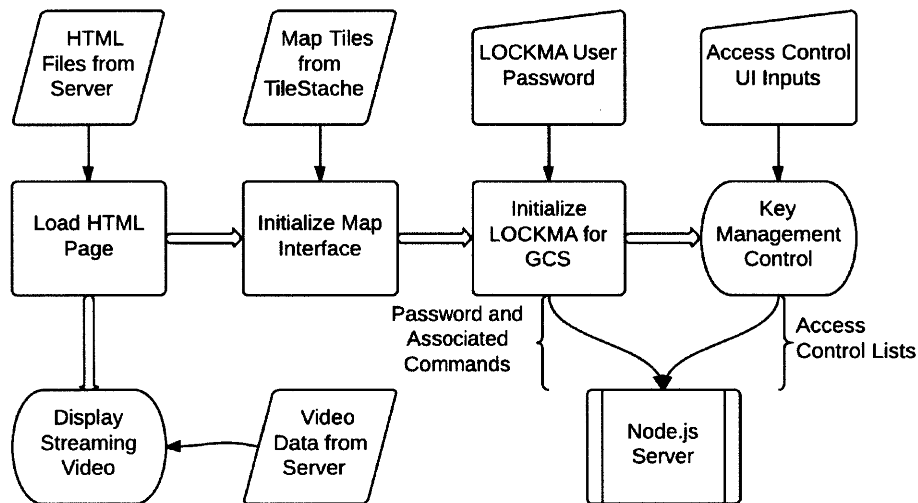


Figure 4-5: This diagram shows the trend of execution, the inputs, and the outputs of the primary access control interface, whose parent process is a web browser. Its primary role is to either accept user commands or contextually synthesize access control lists, and send those to the Key Management Control Center. Because the video streaming is provided by the foundational technology this UI was based off of, it is largely an afterthought in comparison.

The UI is an HTML page with fairly basic CSS formatting. When it is initially loaded, the video stream immediately gets connected through an interface already provided by the node-dronestream project. The map loads, with OpenLayers drawing from the TileStache server automatically, and the initial, pre-specified graphical elements for the interface are drawn to the map based on latitude and longitude coordinates.

The commands issued from the control region on the left side of the screen (pictured in Figure 3-5) are fairly straightforward: they provide info about the intent of the command and any additional values associated; additional values include a password for unlocking LOCKMA (which is only sent over local interfaces, even through the web browser) and the list of receiving GTs when a re-keying command is issued.

Once unlocks are done for both the GCS and UAV, re-keying commands can be sent. These happen whenever the UI's reference of which machines should receive keys changes. When set to manual mode, this is fairly straightforward: if a checkbox changes values, a re-key is sent. When in automatic mode, the map is utilized. The map interface is set up to respond to mouse clicks. For the sake of the emulation, the virtual UAV follows the shortest realistic path to a clicked target for an airplane with a predetermined velocity and turning radius. Then, the access control list is updated based on graphical collisions of the virtual UAV and the specified regions of influence. Whenever the ACL changes, a re-keying command is sent through the server. The main loop, operating at 33 frames per second, is as follows:

```
if (goalSet && Math.abs(currentY - goalPoint[1]) < 50 &&
    Math.abs(currentX - goalPoint[0]) < 50) {
  return;
}
var newDirection = 180 / Math.PI *
                   Math.atan2(goalPoint[1] - currentY,
                              goalPoint[0] - currentX);
if (goalSet && currentDirection != newDirection) {
  var goalDirection = (newDirection - currentDirection +
                       720) % 360
  if (goalDirection < 180) {
    if (goalDirection < ROTATIONAL_VEL)
      rotateSymbol(goalDirection);
    else
      rotateSymbol(ROTATIONAL_VEL);
```

```
  }
  else {
    if (goalDirection > 360 - ROTATIONAL_VEL)
      rotateSymbol(goalDirection);
    else
      rotateSymbol(-ROTATIONAL_VEL);
  }
}
currentY += Math.sin(currentDirection * RAD_CONV) *
            VEL_PER_MS * 30;
currentX += Math.cos(currentDirection * RAD_CONV) *
            VEL_PER_MS * 30;

var coords = latLong(currentX, currentY);

uav.move(coords);
map.panTo(coords);

currentPerms = checkCollisions();
if (!currentPerms.plainEquals(previousPerms)) {
  updateKeys();
}
```

The above code calculates the exact latitude/longitude coordinates and angle to which the virtual UAV should move after 30 milliseconds, moves and rotates the virtual UAV on the displayed map to indicate the change, then sees if the list of collisions is different from that of the previous frame. If it is, a re-keying message is sent if the interface is set to automatic mode.

JavaScript does not have true concurrency within a single web page, so change in the "goal point" can be caused by a registered click event that operates during the 30 millisecond break between frames. The operation cycle of the map interface does not

incur sufficient computational load to cause lag in the video playback.

Feature Issue

In the earliest stage of development, Google Maps was used for the map interface of the project. A key requirement of the project is that it needs to be executable in an isolated space, entirely offline, and the Google Maps API is online-only, disallowing users from creating a local copy. Many paths were followed to attempt to get an offline version of Google Maps including, somewhat interestingly, an Android component that Google was developing for exactly such offline use. In the end, it was determined to be impossible for PCs to do this even for academic use.

Switching to OpenLayers not only solved this problem, but provided graphical manipulation capabilities that in retrospect were even easier to utilize than Google Maps' features, resulting in the cohesive virtual movement the project features today.

4.3.3 Key Management Control Center

The architecture for this process is found in Figure 4-6.

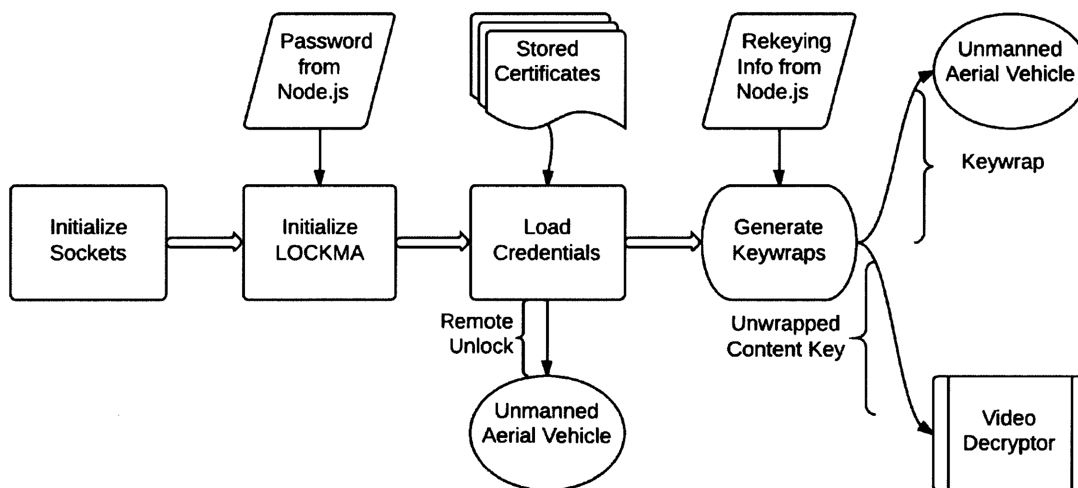


Figure 4-6: This diagram shows the trend of execution, the inputs, and the outputs of the KM Control Center process. Its main function is that when prompted by the Node.js server, it generates a new keywrap to cryptographically enforce the specified access control permissions.

The Control Center on the GCS is where keys originate from. In reality, startup looks very much like on the UAV's KM Process. The Control Center first sets up a UDP socket to the UAV and a TCP socket to each of the Node.js server and the Video Decryptor. Then, it simply waits on unlock commands for itself, which requires a valid password, and then the UAV, so encryption keys can be changed.

The two unique scenes in the Control Center are loading client certificates and newly encrypting keys. The following is how it loads certificates for the clients, rather than merely its own device certificate:

```
uint8 cert_data[1024];
uint8 *cert_data_ptr = cert_data;
uint32 data_len = 1024;

read_file( file , &cert_data_ptr , &data_len );
lockma_cache_recipient_credential( &lockma, *index ,
                                   cert_data_ptr , data_len );
(*index)++;
```

As expected, it is very similar to loading its own certificates, but it applies to the key agreement certificates only and allows data to be encrypted to them so that only specified recipients may read it. In Key Management, this feature is exclusively used to distribute symmetric keys, as PKE is more costly for large packets than symmetric-key encryption.

Generating and encrypting a new key before sending the keywrap to the UAV is done as follows:

```
lockma_application_package_create( &lockma,
  KEY_PACKAGE_DEMO_META, sizeof(KEY_PACKAGE_DEMO_META) ,
  &app_pkg );

key_meta_data[0] = (uint8) channel_nbr;
app_rng_fn( key_meta_data + KEY_META_CHANNEL_LEN,
```

```
            KEY_META_AES_CTR_IV_LEN );
cur_time = htonl( (int32) time(NULL) );
memcpy( key_meta_data + KEY_META_CHANNEL_LEN +
        KEY_META_AES_CTR_IV_LEN, &cur_time,
        sizeof(cur_time) );

lockma_application_package_add_key_meta( &lockma,
                key_meta_data, KEY_META_LEN, app_pkg );

lockma_generate_application_package( &lockma,
                keywrap_slot_idx, app_pkg );

lockma_add_recipient( &lockma, keywrap_slot_idx, //for UAV
    recipient_cache_idx, keywrap_recipient_idx, NULL, 0);
keywrap_recipient_idx++;

for (i = 0; i < num_gts && i < MAX_GTS; i++) {
  if (gts_allowed(i)) {
    lockma_add_recipient( &lockma, keywrap_slot_idx,
            i + 1, keywrap_recipient_idx, NULL, 0 );
    keywrap_recipient_idx++;
  }
}

km_msg_data = NULL;
km_msg_data_len = 0;
lockma_get_signing_credential( &lockma, &km_msg_data,
                               &km_msg_data_len, KM_FALSE );

keywrap_data = NULL;
```

```
keywrap_data_len = 0;
lockma_get_keywrap( &lockma, keywrap_slot_idx, &keywrap_data,
                    &keywrap_data_len, KM_FALSE);
```

LOCKMA must first create a package for the keywrap, then add a key, generate the package, and then add recipients. We always add the UAV as a recipient, then we add the Ground Terminals that are in the current Access Control List that was sent to us by the Node.js server. After that, we get the signing credential and keywrap from the application package, which we can later send to the UAV for distribution.

4.3.4 Video Decryptor Process

The architecture for this process is found in Figure 4-7.

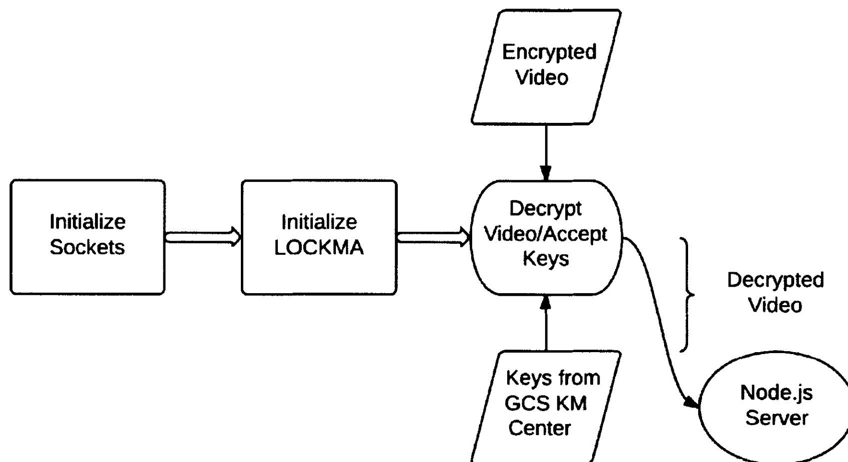


Figure 4-7: This diagram shows the trend of execution, the inputs, and the outputs of the Video Decryptor process. As expected, its functionality almost perfectly mirrors that of the Video Encryptor.

The Video Decryptor process is remarkably symmetric to the Encryptor on the UAV. Its UDP socket is an input, and its second TCP socket, to the Node.js server, is its output, and it performs decryptions instead of encryptions on the data. The only other noticeable difference is that it does not need to set a delay on "switching" keys—LOCKMA channel number is specified by an encrypted packet, so the Decryptor can

merely store an array of the most recent keys at once, so that it will definitely be ready for decryption by the time the Encryptor's delay is finished and the new key enters usage. The channel number for a key is specified by the KM Center on creation, so it will always be the same between the Encryptor and Decryptor.

As expected, Video Decryptor establishes its sockets on startup, performs the necessary subset of LOCKMA administrative actions, and enters its terminal loop. On each iteration of the loop, it waits both for a key from the KM Center and for a video frame from the UAV. After extracting the header data specifying channel number from a received video frame, the relevant decryption command is:

```
lockma_decrypt_app_data_aes_gcm(
    aes_core.channels[active_channel_nbr].key, // key
    (const uint8 *) &sequence_nbr,   // iv address
    sizeof(uint32),                  // iv length
    packet_ptr,                      // ciphertext address
    (uint32) payload_len,            // ciphertext length
    (const uint8 *) auth_data,       // auth data address
    sizeof(auth_data),               // auth data length
    icv_ptr,                         // Integrity Check Value
    plaintext_ptr,                   // plaintext address
    plaintext_len );                 // plaintext length
```

The video frame as it originally came from the UAV's native output can now be found at plaintext_ptr, where it gets shipped off to the Node.js server for streaming.

4.4 Android Ground Terminal

The GTs are actually running a single application, albeit utilizing multiple threads (as a requirement enforced by Android). As such, we provide the high-level architecture for this application here, instead of in a later section. The architecture is found in Figure 4-8.

As stated in Chapter 2, this was based on the Parrot AR.Freeflight Android ap-

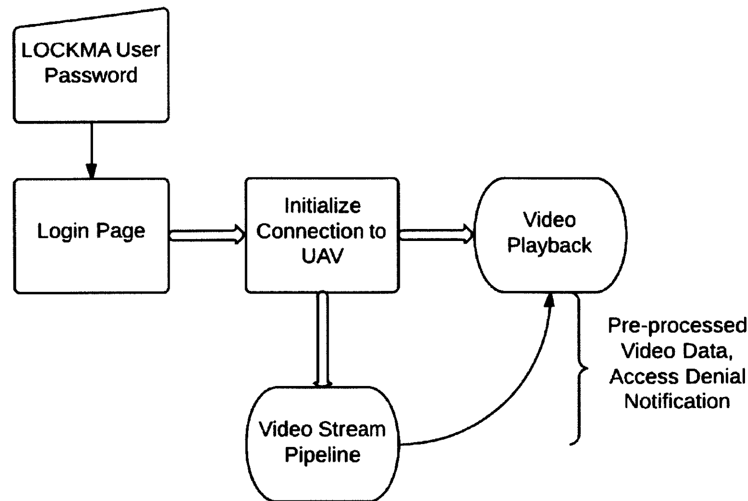


Figure 4-8: This diagram shows the trend of execution, the inputs, and the outputs of the Ground Terminal application as a whole. It mostly consists of a rendering thread and a video processing thread, which run after providing one's LOCKMA user password. The pipeline thread is discussed in more depth later in this section.

plication. As such, more of our code structure was enforced upon us than on the other two platforms. The Android devices we used were Asus Eee Pad Transformer Primes, but the Android SDK's build system meant that our packages were runnable on any Android machine running 4.1.0 or later.

Using the Android NDK for C code was one of the struggles of the project, as documentation is relatively opaque. However, once understood, the steps for inclusion are fairly straightforward, and will be discussed as part of the Key Management and Encryption subsection. The LOCKMA code itself proved very easy to add to the package, once the library was properly compiled and linked.

4.4.1 Android Application UI

The first stage of the UI simply presents a password entry screen with a button to proceed. The code used to override the default Android password behavior, in order to mask all characters properly rather than leaving the last character visible, can be found in Appendix A.1. The "Dashboard" screen has otherwise been cleared.

When logged in, the user is taken to the terminal activity, which features the expected video playback. This activity features several sub-components that perform the real graphical manipulations seen on screen, most notably the HudViewController, which is where our changes were generally directed. This controller's VideoStageView featured the actual acceptance of the video stream, which we had no reason to change at the Java level for this project.

Due to the inability for the primary UI thread to perform network operations, we spawned a new thread for other visual manipulations we wanted, which are based in part on communication between the Java code and the native C code running the video pipeline. On start up, this new thread detects the IP address of the GT to determine how to color-code the bar at the bottom of the device, using a class method of the HudViewController to place the correctly-colored bar at the bottom.

Then, using a basic Java socket and waiting for input in its own independent thread, this extra class also causes the HudViewController to display or remove the "ACCESS DENIED" message from the screen. All sprites involved get scaled by conveniently-provided methods in the Sprite class to fit the Android screen they find themselves on.

4.4.2 Video Stream Pipeline

The architecture for this thread is found in Figure 4-9.

As stated, Android disallows network operations in the primary UI thread. Of course, due to the nature of requiring a video stream persistently posted to a screen with other asynchronous commands in use throughout the original application, the video pipeline utilized by Freeflight needs its own thread regardless.

The pipeline spawns from the entry point of the native code, which is invoked by a service acting as a sub-component of the terminal activity. The primary change here was removing additional extraneous threads from the process, which was done more so for debugging purposes than for speed or application size. However, it's true that in the end, the application is actually much smaller with LOCKMA as part of this project than with its original feature set without LOCKMA.

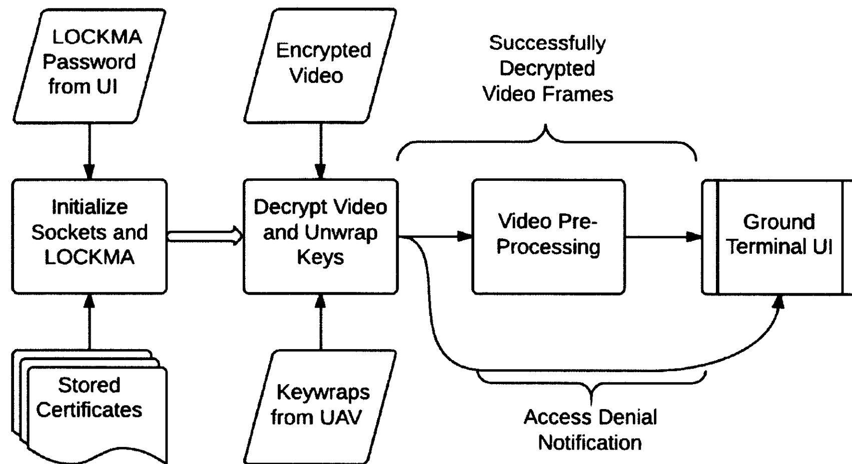


Figure 4-9: This diagram shows the trend of execution, the inputs, and the outputs of the Video Pipeline, which runs as a sub-component of the Ground Terminal application. Unlike those applications that we developed ourselves, the pipeline nature of this means that no point of execution is exactly terminal, though all but the first activity could be considered as such. Data is transferred from activity to activity until FFMPEG eventually delivers it to the GT User Interface.

Another change was required for behavior we wanted in the application. Namely, if video frames were not detected near the beginning of the pipeline, the application would try to force the drone to reconnect to the Android device using a pre-specified communication packet that tells the drone to drop all of its current connections and reconnect to the sending machine. By this method, any device can actually in theory wrest control of the drone (remember, it is actually a toy) from any other party. We wanted the Video Encryptor to maintain control of the video stream, so this feature was suppressed entirely. If at some stage of the pipeline a size of zero is detected, the pipeline starts over from receiving a video packet.

Lastly, we changed the act of waiting on video frame input to utilize our select idiom, waiting on both the video frame and key management messages, rather than only the video frame.

Capability Issue

We mentioned in Chapter 3 that an earlier iteration of the demo featured GTs that were PCs. One key reason for this was that there was an incredibly hard-to-

track bug that presented itself when we switched the video pipeline from accepting on TCP sockets to using UDP. Due to the nature and difficulty of debugging C code, especially in threads not directly spawned by the Java Virtual Machine, this was incredibly difficult to track down. The bug presented as a SIGBUS error at memory address 0, which would normally be a segmentation fault.

Eventually, it was found to be a complex socket error triggered from within the FFMPEG component that Freeflight uses to actually render H264 video frames (which are the format the drone streams as). Further digging showed that this socket error was a fairly ordinary issue that should merely trigger a reconnect attempt, but the Android firmware we had at the time had a major bug that could not handle failed writes to TCP sockets under certain conditions. After a dramatic series of events, the bug was finally resolved by, indeed, a firmware update.

While tracking down the source of this issue, several interesting debugging techniques were discovered for working simultaneously with C and Java code. Although, most of them proved to only be useful from Java-spawned threads, and not from threads spawned within native C code.

4.4.3 Key Management and Encryption

For this final piece, we effectively merged both the Video Decryptor from the GCS and the KM Process from the UAV, and we instrumented a single file with all of it, at the point in the existing C code accepted video frame data from the UAV. At every stage after this, certain aspects of the header are checked and analyzed for consistency, so there was no real option to postpone the decryption to the next stage unless we wanted to risk breaking the existing streaming functionality.

It proved to be no real hurdle. LOCKMA initialization, including certificate reading thanks to the password being input from the first UI screen, was done the first time the pipeline was entered; from then, both key management and decryption happened when appropriate messages were received. The only real hassle was combining network calls into their triggered function callbacks, but fundamentally, the code was already there. Comparatively speaking, it was more work to find the relevant "socket

entry point", back in the early stages of the project when the Android NDK was less familiar, than it was to add LOCKMA instrumentation to the code itself.

Building LOCKMA for Android

While instrumenting code with LOCKMA calls proved as uneventful expected, the build system was not so simple. The Android NDK is not the easiest tool to navigate. Compiling LOCKMA for ARM using the correct processor was the most difficult step. For most platforms, LOCKMA will be built locally and CMake can find the right toolchain based on machine. Android projects, on the other hand, get built for Android by other systems, and have a specialized toolchain for making libraries that work on ARM for the Android OS. Thus, the real issue we found was with the CMake system, which would still build x86 LOCKMA even when the Android NDK was being told to compile it through its normal channels.

Luckily, the Android CMake project, noted in Appendix A.3, solved the issue for us. With it, we simply generated a new platform-appropriate toolchain that CMake could use more easily. The best part is that it was easily revertible, as setting CMake to re-initialize would go back to re-using x86 Linux defaults. LOCKMA got built as a static library for ARM, our project's include files were placed in the existing path for the Android project, and linking LOCKMA with the NDK followed the same route as with any other static library. No further issues occurred, and the default build settings generated ARM code that was empirically sufficiently optimal for video playback to not experience an observable delay.

4.5 Implementation Conclusion

In this chapter, we discussed the implementation details of the project and the issues the project faced. Reading this chapter should have provided enlightenment on how to make use of the LOCKMA library to provide fast, usable cryptography to an existing application, most notably for a new access control system in line with the one that we have built. We also went into the detail necessary for understanding certain build environments, namely CMake and the Android SDK and NDK, that

may have previously been unfamiliar. Most directly, we conveyed an idea of how the demo fully came together into the form it holds today. The result of this project was a great success, streaming live video in a way that does not incur a more noticeable delay than the existing network delay, and that never introduced any extended lag from operations leaking into future frame processing time.

Chapter 5

Conclusions

The principal contributions of this thesis focused on developing a usable access control system for active missions featuring multiple parties needing access to a data stream, demonstrating an example use case while encouraging the use of access control in more general systems, and demonstrating the usability of LOCKMA as a component and its usefulness in developing this type of usable access control. This thesis determined that LOCKMA's use is indeed sufficient for these purposes, and should serve as a useful tool for those wanting to build security-conscious applications, especially access control systems, with LOCKMA in the future.

Here, we document more precise results relating to the overhead our system introduced, and also discuss other projects that are in-progress or being discussed for the future. Then, we discuss what the author has learned in the process of developing this work.

5.1 Technical Results of LOCKMA

LOCKMA proved to be both a small, fast, and easy-to-use component. When compiled both for ARM and x86, the size of the library component never exceeded 4 megabytes. The version compiled with Microsoft Visual Studio for x86 Windows was 2.5 MB. However, 4 MB is quite small enough even to put on an embedded platform. As stated previously, the APK of our Freeflight modification, which had a few extra-

neous features cut, was still smaller with LOCKMA added for both Key Management and decryption of the video stream.

For speed, there were no human-observable issues; the inevitable delay incurred by encryption and decryption was entirely overshadowed by the basic network delay in streaming unencrypted video from the same source, when humans were comparing them. In practice, for reference, the largest delay incurred by decryption was approximately 16 milliseconds on the 1.4 GHz processor of the ASUS Transformer Prime, a much slower processor than can be expected of a modern PC, for the largest frames of over 20 kilobytes. This is compared to the expected break between frames of 33 ms, when streamed at 30 frames per second; it is a greater percentage of an attempt at streaming 60 frames per second, but there are two counterpoints:

- 60 fps would also be that much more taxing to the Android on its own, without security. A more impressive GPU could admittedly serve to lessen that issue, but the entire pipeline structure provided would still be doing twice as much work.
- Most H264 frames are smaller than 20 kilobytes. Only reference frames are so taxing; over 90% of the frames were under 7 kilobytes and were decrypted in under 6 milliseconds, thus allowing the machine to catch up on frames very quickly even if it were to fall behind.

On the PC, more variance but lower averages were found. The largest time taken was again found to be about 16 milliseconds for a 22 kilobyte frame, but the majority of the smaller frames were sub-millisecond. A scatterplot of PC performance of LOCKMA's symmetric-key decryption is found in Figure 5-1.

Even with this degree of variance, the majority of frames are small enough that decryption is incredibly fast on average. We found the average on the PC to be under 1.5 milliseconds consistently (1.35 milliseconds from the set used for Figure 5-1), and H264 streams are structured such that you will never have an unexpected number of consecutive large frames in a row, so even probabilistic drifting becomes impossible. Encryption and decryption in counter mode both use the same direction of AES,

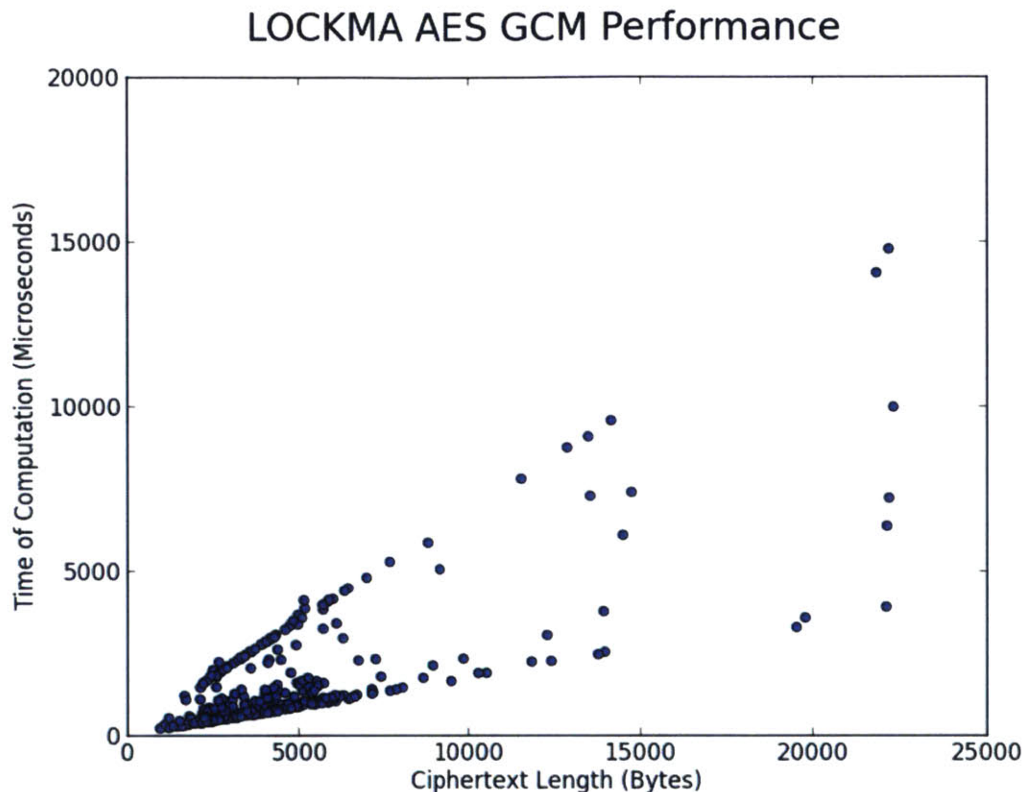


Figure 5-1: LOCKMA AES GCM decryption was measured on an x86 PC with a 2.3 GHz processor. No hardware acceleration was available on the system. CTR-mode encryption and decryption is parallelizable, but there are finitely many CPUs available. As such, the asymptotic growth of the encryption/decryption function is linear on average. The potential range of run times also grows linearly, which means that the function's time taken has nontrivial variance when encrypting or decrypting sufficiently much data at once.

and all Message Authentication Codes (MACs) are unidirectional; thus, the speed of encryption and decryption in AES GCM should always be the same when operating on the same platform.

We conclude that LOCKMA will not bottleneck for any comparable video streaming on a reasonably modern platform.

Asymmetric encryption happens much less frequently (exclusively for key management) and, as shown, can also be handled as a separate process entirely. Thus, its speed is not strikingly relevant, though it is worth noting that it did not incur human-noticeable delays in the Android application's video playback, in which keys

were changed in the same thread as video was decrypted.

For CPU utilization, there were also no observable issues. The UAV, whose hardware exists almost exclusively to be useful in being flown and relaying video frames, and which certainly had the slowest processor (at 1 GHz and single-core), was still not taxed even with two LOCKMA programs running on it. Load average for all steps in streaming and encrypting video averaged 0.04. Load did spike as keywraps were handled and decrypted, but even when we overloaded it with several keywraps per second, the processor did not report more than 0.45 short-term load average; over a complete run, medium-term load average never exceeded 0.08.

On the PC, LOCKMA-utilizing processes barely registered as even a blip on the CPU radar, occasionally hitting 1% CPU utilization. Video rendering in the Chrome browser was comparatively taxing.

Lastly and perhaps most importantly, my experience with LOCKMA showed me how easy it is to augment existing code with it. There may be a non-trivial number of function calls required to get Key Management fully functional, but those calls are very symmetric across a given project, and LOCKMA is already usefully configured to compile easily on both Windows and Linux. It was more work to integrate it into Android, due to the toolchain issue and x86 Linux wanting to configure CMake for running on its own platform by default, but this was already a solved issue.

5.2 Future Work

There are two access control augmentations that will allow us to leverage LOCKMA's key management capabilities to demonstrate even more powerful systems. One involves a minor change to our existing project that is a strict improvement in its current capabilities. This is the act of being able to add brand-new GTs to a mission that is already in-progress. The other, while it is certainly useful for access control systems, is more geared as a demonstration of the generality of LOCKMA's key management techniques. This would be a very major change to our existing demo, and would involve the removal of all application-specific encryption and decryption from

the projects: we could use IPsec, the Internet Protocol security layer, for encryption and decryption of communication between machines, and simply use LOCKMA to control the relevant IPsec keys.

5.2.1 Dynamically Adding New Ground Terminals

This is a very direct augmentation to the current demo that is possible. Namely, by allowing the GCS to accept new key agreement credentials over the network and adding an interface for dynamically adding new color-coded regions to the OpenLayers map interface, we can implement a system for an arbitrary number of potential Ground Terminals to be added to the demonstration even after it has started up. Not only is this a strict increase in the versatility of the project, but it also fits into our original motivational model of the DEA chase scene—a suspect or culprit could travel to any number of regions under any number of jurisdictions, and it would certainly be useful to give the relevant police departments all available information for their efficacy in the case to be maximized.

This requires a non-trivial upgrade in the systems we implemented for the GCS, especially for the GUI, but for the GTs, the change is fairly minimal and occurs exclusively in start-up.

5.2.2 LOCKMA for Key Management, IPsec for Encryption

Presently, LOCKMA is being used for application-specific encryption, as happens with TLS. However, it has the capability to interact directly with IPsec, the Internet Protocol security suite, which is an open standard specified by IETF RFC 4301 [17]. There is some amount of multicast support [18], but it is not required to be implemented, and even when it is, it can be difficult to work with. However, IPsec has an invaluable feature, the ability to secure all communication between two systems or groups of systems, and there are certainly use cases where LOCKMA's available key management techniques are more desirable than the pairwise networked key agreements required by IPsec currently for key management.

Linux implements IPsec in its kernel, and has a useful forward-facing interface for managing keys locally. With key distribution handled through LOCKMA, a group of computers could easily set up IPsec to encrypt their traffic at the Internet Protocol level, for example in a multi-party video conference using non-free (or non-open-source) software that could not already usefully be instrumented with LOCKMA at the application level. And once groups have this set up, it would work just as well with any application, and the only new setup required would be for new machines or groups of machines, rather than requiring a new machine to add LOCKMA separately for every application it uses, which is presently what SSL/TLS has to do. Some work has already been done in this area, but it is as of yet unfinished and untested. In theory, however, the only requirement in Linux is interfacing existing LOCKMA key management code with the IPsec command-line interface as the superuser.

5.3 Learning Experience

Working on this thesis has been one of the most thorough learning experiences of my life. I have gained both technical knowledge and a deeper understanding of what it means to work in computer security and on long-term projects.

5.3.1 New Technologies and Algorithms

Through research for this thesis, I have worked with a variety of tools and systems for the first time in my life. Some of these have, in the meanwhile, already been used in other aspects of my programming career. Others I am sure will serve me well in the future.

I had never programmed for Android before, and in fact had never even utilized Java Native Interface (JNI), which is how C/C++ code is utilized from Java. Working with a pre-constructed project made the Java and User Interface side fairly straightforward throughout my work, but the JNI and Android NDK left no shortage of hair-pulling when I was working with the C side of the project, in which video data was handled. This came even before instrumenting the code with LOCKMA; as

stated in Chapter 4, the largest issue I dealt with ended up being solvable only by a firmware update (or else completely overhauling the well-defined FFMPEG library), but while attempting to debug this issue myself, I learned a variety of interesting debugging techniques and features in the JNI that I would not have even needed in order to complete the project as it exists today. Of course, I also learned how to link new libraries into an Android project through the NDK, as part of adding LOCKMA to it.

My JavaScript experience was also severely limited prior to work on this project. I had worked with single-page HTML/JavaScript in the past, but this required me to not only render dynamic content with JavaScript through networking, but also resulted in developing server-side code with Node.js, which was quite the fun adventure. It has heavily motivated me to invest more time in studying efficient multi-user servers, which goes beyond the single-user server acting as a process go-between for the sake of this demonstration.

LOCKMA itself, of course, was incredibly novel to work with. When I initially began working with it, I was sadly skeptical that it could be so simple to instrument code with its cryptography; in reality, it was just as simple to use then as it is now, but it was poorly-documented when I was first introduced. It did not take long for me to recognize that it merely needed its documentation to be written down properly to become the tool it was meant to be. Compared to something like OpenSSL, utilizing LOCKMA feels much easier. Though it's true that at present, LOCKMA serves a different need in the first place. But at this point, I feel like I could guide anyone to utilizing LOCKMA in their application.

Finally, my experience in this project has also caused me to look further into Elliptic Curve Cryptography than I had considered previously. Around the middle of my work on this project, I was taking an applied computer security class. The final project was fairly open-ended, but one of the suggestions and, as a result, most commonly-executed project was an "encrypted file system". For a multi-user system, it is obvious that it would require some degree of public-key cryptography for the sake of preventing universal user access. Everyone else in the class that went with

this project defaulted to RSA. Using ECC, our PKE had three clear advantages over RSA-using systems. Notably, these came from the fact that ECC private keys with modern systems can be arbitrary bit strings, and the safety involved comes from the curve used. This is contrasted with RSA, in which you need to pick two large primes that when multiplied cannot be factored easily by any kind of inference (must be sufficiently large and sufficiently far apart, etc.), and as such the RSA keys can really only be effectively represented as numerical fields. The advantages we found are as follows:

- Generation of many RSA keys is inherently going to be less safe over time. You limit the number of primes near the size of the key you are trying to generate. Anyone not incredibly well-versed in the relevant mathematics concerning primes will have a hard time even verifying that a key generator is going to be safe when used. ECC keys, being allowed as arbitrary bit strings, can effectively just be randomly generated, as with AES keys, so only the safety of the RNG matters.

- ECC private keys are more versatile. Through the generation of a single private key that corresponds to a certain public key, you can trivially make derivative private keys by appending bits to the original private key. The resulting public keys do not have an obvious correlation.

- ECC keys require less storage than RSA keys of the same security factor, which makes them capable of being more ubiquitous throughout the project. It's not merely about the raw number of users; through our derivative private keys, we made use of derivative public keys throughout files in order to create anonymized signatures to signal administrative permissions to files through our virtual file server. Even attempting something similar would have been significantly more costly with RSA keys, if it were possible without compromising the security of the RSA system.

As such, while I had been introduced to the world of computer security prior to this

project, I believe my introduction to elliptic curves heavily opened my eyes as to the potential of modern public-key cryptography.

5.3.2 Non-technical Lessons

While there is less detail to go into on this subject, my non-technical lessons over the course of this project were arguably more valuable to me as a computer scientist and as a person.

The largest issue I had—which was fixed with an Android firmware update—was an intermittent socket error. All told, it gave me about eight months of intermittent headache. My time estimates on how long it would take me to fix the error were always wrong, right up until the day I forced the update that fixed it. I was still productive at points during this, for a couple months dropping Android entirely to get a PC GT up and running, but the way I handled this after the initial wave of debugging—which for a while appeared to be a success—simply was not correct. The realistic time estimate for debugging when no debugging output I could get from built-in Android tools appeared relevant was always "at least a few weeks" rather than "I think I'll have it next Monday", but more importantly I should have asked for help sooner. Even though it turns out there was no one I was working near that was more familiar with the actual set-up I had—accepting signals from within natively-spawned threads through the Android NDK—it was discussing the issue with people that eventually led me to discovering that one of the software libraries being used actually had a compatibility issue with older versions of the Android OS, thus leading to the discovery of both the source of the error and the fix.

The other most notable lesson for me was how to manage my own work on a long-term project. Before this work, I was still too used to school-sized projects that at most took a few weeks of hardcore code diving. I had worked in a medium-sized team once before developing a game consisting of a relatively sizeable code base; however, even that was only 8 weeks long and further consisted of a thorough structure imposed upon us. When I began this project, I very much dove in with the school mindset of "just code everything as I go", since it was a project that I was mostly able to

work on alone, though I had technical guidance from Dan Utin and worked directly with another lab employee for compiling ARM binaries for the drone. As a result, my organization was very poor, though in the early stages of the project, my output somehow managed to keep up with expectations. As the project progressed, however, and especially as the Android bug became more of an immediate issue, I had to figure out more efficient ways to manage my time in order to not completely halt while the bug itself was unsolved.

I began to develop a system in which I dynamically evaluated how much time a halting task deserved so I could maintain enough time per day working on tasks that were moving along more smoothly. In this way, I managed to avoid losing inertia for tasks that could have taken much longer had I spent entire days focused only on debugging one issue. Further, even having such issues at the back of my mind motivated me to come up with more detailed plans for implementation timelines. By the end of the project, I was able to much more closely keep to my ideal schedule in putting the finishing touches on the Android Ground Terminals. Even now, I find that setting smaller goals for myself in writing this thesis step by step has causes me to be much more productive and efficient than I originally expected of myself, as writing has always been an aversion of mine.

Appendix A

Utilized Non-Critical Components

This chapter documents sources that were utilized but not warranted as being described as whole project dependencies for their technical contributions in Chapter 2.

A.1 Proper Android Password Masking

This code was provided on StackOverflow from user chipiik for question 6360222.

```
private class HiddenPassTransformationMethod implements
    TransformationMethod {
  private char DOT = '\u2022';

  @Override
  public CharSequence getTransformation(
      final CharSequence charSequence, final View view) {
    return new PassCharSequence(charSequence);
  }

  @Override
  public void onFocusChanged(final View view,
        final CharSequence charSequence, final boolean b,
```

```java
        final int i, final Rect rect) {
    //nothing to do here
  }

  private class PassCharSequence implements CharSequence {

    private final CharSequence charSequence;

    public PassCharSequence(final CharSequence charSequence) {
      this.charSequence = charSequence;
    }

    @Override
    public char charAt(final int index) {
      return DOT;
    }

    @Override
    public int length() {
      return charSequence.length();
    }

    @Override
    public CharSequence subSequence(final int start,
                                    final int end) {
      return new PassCharSequence(
                  charSequence.subSequence(start, end));
    }
  }
}
```

A.2 iPhone-Style Auto/Manual Switch

For the switch viewable in our Ground Control Station User Interface, we utilized the iOS Checkboxes project, free for academic use, and available at
http://ios-checkboxes.awardwinningfjords.com.
It is fairly straightforward to add and utilize in any web-based project.

A.3 Android CMake

The Android CMake project ended up greatly streamlining the build process for making LOCKMA work correctly for Android applications. Once the correct toolchain is integrated into the Makefiles through CMake, the built library can be added to the JNI for an application the same way as any other static library. This system can be found at
https://code.google.com/p/android-cmake/.

Bibliography

[1] "Parrot AR Drone 2.0 website." http://ardrone2.parrot.com/. Accessed: 2014-09-02.

[2] "Android Operating System website." http://www.android.com/. Accessed: 2014-09-02.

[3] R. Khazan and D. Utin, "Lincoln Open Cryptographic Key Management Architecture." *MIT Lincoln Laboratory Tech Notes*, http://www.ll.mit.edu/publications/technotes/TechNote_LOCKMA.pdf, 2012.

[4] R. Khazan and D. Utin, "LOCKMA: Lincoln Open Cryptographic Key Management Architecture." *Cyber Security Division Transition to Practice Technology Guide*, http://www.dhs.gov/sites/default/files/publications/csd-ttp-technology-guide-volume-2.pdf, Page 17, Dec 2013.

[5] H. Stuart, "Drone List Released By FAA Shows Which Police Departments Want To Fly Unmanned Aerial Vehicles." *The Huffington Post*, http://www.huffingtonpost.com/2013/02/08/drone-list-domestic-police-law-enforcement-surveillance_n_2647530.html, Feb 2013.

[6] R. Bowman, "Under Orders From Congress, FAA Makes Way For Commercial Drones." *Forbes*, http://www.forbes.com/sites/robertbowman/2014/07/22/under-orders-from-congress-faa-makes-way-for-commercial-drones/, July 2014.

[7] G. Greenwald, "XKeyscore: NSA tool collects 'nearly everything a user does on the internet'." *The Guardian*, http://www.theguardian.com/world/2013/jul/31/nsa-top-secret-program-online-data, July 2013.

[8] "OpenLayers website." http://openlayers.org/. Accessed: 2014-09-02.

[9] A. Petcher, R. Khazan, and D. Utin, "A Usable Interface for Location-Based Access Control and Over-The-Air Keying in Tactical Environments." Military Communications Conference, Nov 2011.

[10] "Node.js website." http://nodejs.org/. Accessed: 2014-09-02.

[11] "Github page for node-dronestream." https://github.com/bkw/node-dronestream. Accessed: 2014-09-02.

[12] "Google Maps API documentation." https://developers.google.com/maps/documentation/javascript/tutorial#Loading_the_Maps_API. Accessed: 2014-09-02.

[13] "TileStache website." http://tilestache.org/. Accessed: 2014-09-02.

[14] "TileMill, by Mapbox." https://www.mapbox.com/tilemill/. Accessed: 2014-09-02.

[15] "Android Software Development Kit." http://developer.android.com/sdk/index.html. Accessed: 2014-09-02.

[16] "AR Drone SDK version 2.0.1." https://projects.ardrone.org/projects/show/ardrone-api. Accessed: 2014-09-02.

[17] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," Dec 2005. RFC 4301.

[18] B. Weis, G. Gross, and D. Ignjatic, "Multicast Extensions to the Security Architecture for the Internet Protocol," Nov 2008. RFC 5374.