Joseph Malt

Trusted-Path Keyboard

Computer Science Tripos – Part II

Churchill College

May 17, 2019

Acknowledgements

Dr Markus Kuhn, my supervisor, for all his advice, especially his cryptography expertise.

Prof. John Daugman and **Dr Anil Madhavapeddy**, my overseers, for their advice and encouragement.

Dr John Fawcett, my Director of Studies, for his advice, especially when writing the proposal.

Mike Roe, for his input during the project selection stage.

Peter Rugg and **Thomas Wemyss**, for reading a draft of this dissertation and providing useful feedback.

My parents, **Daniel Malt** and **Prof. Jeannette Littlemore**, for proofreading this dissertation.

Declaration

I, Joseph Malt of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Date: 16$^{\text{th}}$ April 2019

Proforma

Candidate Number:	**2346B**
College:	**Churchill College**
Project Title:	**Trusted-Path Keyboard**
Examination:	**Computer Science Tripos – Part II, 2019**
Word Count:	11975[1]
Final Line Count:	1972
Project Originator:	Dr M. G. Kuhn
Supervisor:	Dr M. G. Kuhn

Original Aims of the Project

To design, construct and evaluate a prototype of a device which provides a trusted, keyboard-based input channel to a Linux PC, preventing hardware keyloggers from intercepting or manipulating the input.

Work Completed

All success criteria have been met. The majority of the work has been completed as described in the project proposal (Appendix C). Some small changes have been made:

- The use of Password Authenticated Key Exchange has been replaced with a password-based key derivation function, to simplify the implementation by reducing the number of cryptographic primitives needed.
- The use of an oscilloscope to perform timing measurements has been replaced with the use of a high frame rate camera, which is simpler and provides adequate precision.

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Special Difficulties

None.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aim

The aim of this project was to develop and evaluate a keyboard which provides an input channel that is secure against hardware keystroke loggers ('keyloggers'). These are devices which can be inserted between a keyboard and the computer to which it is connected in order to surreptitiously record key presses.

1.2 Motivation

In recent years, considerable effort has been put into detecting malicious software that has the potential to leak secret information, with software keyloggers being one such example. However, hardware keyloggers can provide an attacker with the same information, whilst being nearly undetectable in software. Such devices are widely available for less than £50. [17]

Since hardware-based attacks require physical access to the device concerned, they are

Figure 1.1: A KeyGrabber USB keylogger, with 1p coin for scale.

harder to execute than those based on malicious software. Nonetheless, individuals who are at higher risk may be concerned about so-called 'evil maid' attacks [34], in which an attacker with brief physical access performs an undetectable modification which compromises security. Hardware keyloggers are a cheap and quick-to-install way to perform such an attack.

It is difficult to secure a computer system against an attacker who can physically modify it: as Ross Anderson puts it, 'many security mechanisms can be defeated if a bad man has physical access to them'. [2, p.365]

Given an attacker with sufficient resources, this may be true, but this does not mean frustrating such attacks is worthless. Hardware keyloggers can be installed more quickly and cheaply than other hardware-based attacks, so frustrating their use may be sufficient to inhibit some attackers, such as those who only have physical access for a short time.

A good example of where a hardware keylogger might be a concern is an open office: if Alice wants to steal her colleague Bob's password, she can easily attach a keylogger to his PC while he goes out for lunch.

1.3 Existing work

There do not appear to be any existing products on the market that provide an encrypted keyboard channel over a wired connection. However, Bluetooth and other wireless keyboards generally use encryption on the radio link, and are in many ways comparable. An overview of Bluetooth cryptography can be found in [28].

Recent research has looked into the feasibility of detecting hardware keyloggers from software by observing different behaviour, both at the physical layer and higher up in the protocol stack. For some keyloggers, this has been successful [25]. Nonetheless, it is possible to construct a keylogger which listens completely passively (for example by electrically isolating the logging circuitry from the USB lines), and is indistinguishable from a directly-attached keyboard.

1.4 Project overview

In this project I implement two complementary programs, one running on a microcontroller with a keyboard attached and the other running on a Linux PC. These programs communicate over a USB connection to provide an encrypted, authenticated channel over which keystrokes can be securely transmitted from the microcontroller to the host. As part of this I design and implement a cryptographic protocol based on the AES block cipher.

I then analyse the latency of the system to determine to what extent the encrypted channel delays the processing of keystrokes.

Finally I analyse the security of the system against a variety of potential attacks, and compare it to an ordinary keyboard from a usability perspective.

Chapter 2

Preparation

2.1 Starting point

- I have developed simple programs for microcontrollers before, but I have never implemented cryptographic algorithms or a non-trivial communication protocol.
- I have basic knowledge of C from the Part IB course *Programming in C* and from hobby programming.
- Much of the development for this project took place before the Lent term Cryptography course was given, so I had to do some reading ahead. At the time, I had basic knowledge of cryptography, as well as common software vulnerabilities, from the Part IB Security course.

2.2 Terminology

This project consists of two parts, developed together.

The **device** is a microcontroller with a keyboard and potentially other components attached. It is responsible for encrypting sensitive data. In a commercial product, the microcontroller would most likely be integrated into the keyboard housing.

The **host** is an ordinary PC, to which the device is connected. The system as a whole aims to provide a secure way of entering data on the host.

The system consists of two parallel programs (the 'host program' and 'device program'), and a protocol by which they communicate.

2.3 Requirements

The requirements can broadly be split into the categories of *usability* and *security*.

Usability

U1 The system must provide a keyboard-based input channel which is usable as an ordinary keyboard. After a setup phase, the user should be able to forget they are using a specialised input device, i.e. the device should be *transparent*.

U2 The setup procedure, if it requires user input, should consist of simple and easy-to-follow instructions.

Security

S1 The attacker must not be able to determine which keys are pressed on the keyboard. This includes probabilistic information.

S2 The attacker must not be able to modify or reorder keystrokes.

S3 The attacker must not be able to spoof keystrokes, including replaying past keystrokes.

S4 Provided the user follows instructions correctly, the attacker must not be able to trick them into compromising the system.

Specifically excluded requirements

E1 The system does not need to resist denial-of-service attacks, in which an attacker simply stops the keyboard from working at all (for example by cutting the connection or mangling data).

E2 The system does not need to support anything other than US or UK English QWERTY keyboards.

2.4 Threat model

When designing any security-critical system, a clear threat model is needed. This helps enumerate possible vulnerabilities, and also helps bound the scope of any security measures by putting a limit on what the attacker can do.

The threat model for this device is based on a theoretical attacker who has the following capabilities and limitations:

T1 The attacker has full access to all data transmitted over the serial line between the device and the host system. They can therefore eavesdrop on, modify, replay and inject arbitrary traffic in both directions.

T2 The attacker can temporarily disconnect the device and host as often as they wish. The attacker cannot permanently disconnect the device and host, because denial-of-service attacks are out of scope.

T3 The attacker does not have access to the host system beyond what is normally available to a USB peripheral attached to a GNU/Linux system. In particular, they cannot run arbitrary code on the host or device. They may, however, send traffic which exploits vulnerabilities in the host or device software.

T4 The attacker may not exploit pre-existing vulnerabilities in the USB stack, Linux kernel or any other software on the host that the system relies on.

T5 The device and host can share a small amount of secret key material out-of-band. This cannot be captured by eavesdropping on the serial line.

T6 The attacker may perform side-channel attacks such as timing attacks, exploiting imperfections in the implementation of any cryptography.

T7 The attacker may log data for later processing on a more powerful system, unconstrained by the limited computational power available in the small form factor of a keylogger.

T8 The attacker has full knowledge of the design; the system must not depend on 'security by obscurity'.[1]

T9 The attacker cannot physically tamper with the device or the host.

2.5 Design decisions

In light of the requirements, I made some decisions as to how the system should be implemented.

2.5.1 User as a secure channel

As described by Dr Kuhn in the original project suggestion [18], I decided to exploit the fact that the user can act as a secure channel for transmitting key material between the

[1] This idea is known as Kerckhoffs' Principle and dates from 1883. [16]

host and device. Specifically, the user can be shown a (reasonably short) secret value, such as a password, on screen and required to type it in on the keyboard. This secret never travels over the USB connection, so it cannot be discovered by an attacker.

There are many ways to use such a secret to secure a connection. In this project, I use a *password-based key derivation function* to generate cryptographic keys from the password; these are used to encrypt and authenticate the connection. I chose to use this approach because it is possible to construct such a function using the same cryptographic primitive (the AES block cipher) that I use for encryption and authentication. Using a single primitive minimises the size of the program, which is important on a microcontroller with limited storage.

An alternative method is *Password-Authenticated Key Exchange*, in which the communicating parties negotiate a key with one another. They use the password to ensure that they are genuinely talking to one another, and not a 'man-in-the-middle' attacker, who impersonates each party to the other.

Other methods exist which do not require the exchange of a password. For example, encrypted chat platforms such as WhatsApp encourage users to compare a digest of the shared key [38], while HTTPS uses trusted *certificate authorities* to cryptographically vouch for the identity of one or both communicating parties.

Bluetooth, which is widely used for keyboards and other input devices, commonly uses a shared PIN entered or displayed on both devices (a form of PAKE), [28, p.12] although lower-security devices such as headphones often skip authentication entirely.

Status display

As the device relies on the keyboard for password entry, it must be able to operate in a special 'password entry mode' in which anything entered on the keyboard is interpreted as secret material, and not as ordinary keystrokes to be sent to the host. The device must be able to unambiguously indicate when it is in this mode.

For this I attached an LED to the Arduino, which is illuminated only when the device is in password entry mode.

2.5.2 Using an off-the-shelf keyboard and external microcontroller

It was clear from the beginning that a programmable microcontroller would be needed to implement the encrypted communication protocol. Initially I intended to use a keyboard with a built-in microcontroller, but while these do exist, they are only used by a very small community of hobbyists, and as a result there is almost no documentation.

I therefore decided to use an off-the-shelf USB keyboard and an external microcontroller. The keyboard used for development is a *Trust 20623*, available for under £10, [35] but several other USB keyboards (selected arbitrarily out of the filing cabinet in the Intel Lab) have been verified to work.

For the microcontroller I decided to use the widely-used Arduino Uno [4] platform, which consists of an ATMega328P 8-bit microcontroller, mounted on a circuit board with components for programming over USB, and headers for easy interfacing with electronic components.

2.5.3 Using existing USB hardware

The Arduino interfaces with the keyboard via a *USB Host Shield* [3]. This is a commercially available device which acts as a USB host and communicates with the microcontroller using the *Serial Peripheral Interface* (SPI) standard. At the core of this device is a MAX3421E chip [24], which implements both the low-level USB protocol and the analogue electrical operations (voltage level shifting, differential signalling) needed to interact with a USB device.

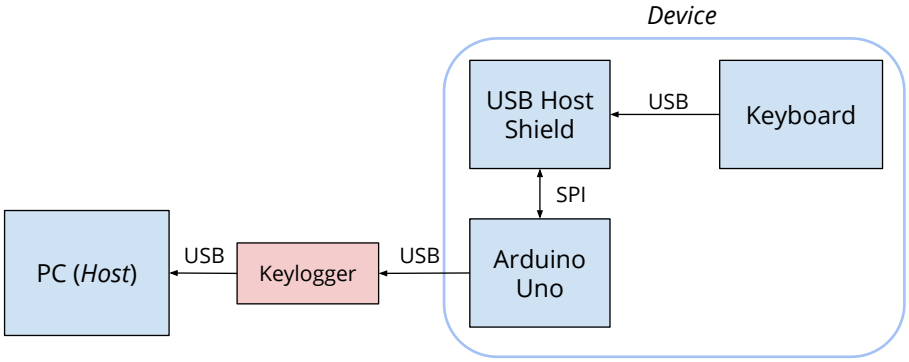

Figure 2.1: Block diagram of the hardware components in the system, including a keylogger if present.

Chapter 3

Implementation

3.1 Hardware

As previously mentioned, the core of the hardware is an ATMega328P microcontroller, on an Arduino Uno development board. Mounted on top of this is the USB Host Shield (the term 'shield' comes from the mounting position), with the MAX3421E USB controller.

Also connected to the Arduino are two small LEDs. One (green) is used as an indicator that the device is in password-entry mode as discussed in subsection 2.5.1, and the other (red) indicates that the password just entered is being processed and the device is not ready for operation. Both LEDs were also used to indicate the program state for debugging during development.[1]

The entire assembly is powered from the USB connection to the host. It is possible to power the Arduino board from a separate power supply, but this is only necessary if components are connected which draw a large current (greater than 100mA).

Figure 3.1 shows the hardware setup.

The ATMega328P, like any microcontroller, is a resource-constrained environment. The chip contains only 2KB of RAM, and operates at a maximum frequency of 20MHz. The non-volatile flash memory is 32KB, which restricts the size of the program. All of these constraints had to be taken into account when designing the system.

[1]The most convenient way to debug an Arduino program is to print data to an attached PC over the serial line; this was not always possible since this project uses the serial line to carry binary messages between the microcontroller and the PC.

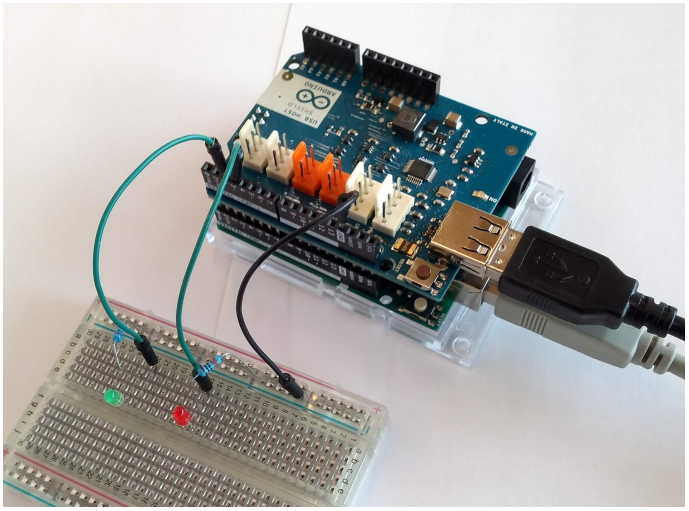

Figure 3.1: The hardware setup of the system, showing the Arduino board with the USB Host Shield mounted on top, and the LEDs on an adjacent breadboard. The white USB cable connects to the host and also supplies power; the black USB cable is from the keyboard.

3.2 Protocol design

The host and device communicate using a custom protocol on top of the USB Abstract Control Model (ACM, sometimes called 'USB Serial') protocol. ACM is designed for emulating hardware serial ports, and as such provides a framework for sending arbitrary sequences of bytes between a peripheral and a host system. It is commonly used for connecting devices such as low-speed modems. [30]

3.2.1 Message structure

The message structure is designed to accommodate both encrypted and plaintext data.

All messages are a fixed length of 32 bytes. Using fixed-length messages simplifies communication as the receiver can detect when a complete message is received simply by waiting for the input buffer to be full.

The first byte is a *magic number* identifying the type of message. This is followed by up to 31 bytes of unencrypted arbitrary data (for all message types except `MSG_ENCRYPTED`) or a 16 byte payload of encrypted data followed by a 15-byte *message authentication code*, as described in subsection 3.3.2. The size of this payload was deliberately chosen to coincide with the size of an AES block, which simplifies encryption and decryption.

The encrypted data inside a `MSG_ENCRYPTED`, when decrypted, has further structure: the first byte is a magic number identifying the type of message (key-down or key-up event), followed by up to 15 bytes of data. Full details can be found in Figure **??**.

List of messages

Name	Magic number (byte 0)	Direction	Purpose	Contents
`MSG_SETUP`	`0xAA`	Device → Host	Set up encrypted communication and transfer IV (counter) for encryption.	Bytes 1-4: initialisation vector for encryption. Bytes 5-31: `0x00`.
`MSG_SETUP_ACK`	`0xBB`	Host → Device	Acknowledge connection setup.	Bytes 1-31: `0x00`.
`MSG_REKEY_INIT`	`0xCC`	Device → Host	Prompt host to generate and display a secret to establish new keys.	Bytes 1-31: `0x00`.
`MSG_ENCRYPTED`	`0xDD`	Device → Host	Wrapper for encrypted data.	Bytes 1-16: Encrypted data. Bytes 17-31: MAC.
`MSG_SALT`	`0xEE`	Host → Device	Salt for key derivation (see section 3.4).	Bytes 1-16: Salt. Bytes 17-31: `0x00`.

Figure 3.2: The messages transmitted between host and device in the system.

List of encrypted messages

All encrypted messages are sent from the device to the host, as the payload of a `MSG_ENCRYPTED`.

Name	Magic number	Purpose	Contents
`EVENT_KEYDOWN`	`0x11`	Key down event from the keyboard.	Byte 1: keycode. Byte 2: modifiers.
`EVENT_KEYUP`	`0x22`	Key up event from the keyboard.	Byte 1: keycode. Byte 2: modifier.

Figure 3.3: The encrypted messages used in the system. N.B. See subsection 3.5.2 for details of keycode and modifier bytes.

3.2.2 State machines

Both the host and device are implemented as *finite state machines*. This is a conceptual model in which at any given time the system is in one of a finite set of states. The current state controls the behaviour of the system, and the system can be made to change state by internal or external triggers. State machines are a common design approach for modelling distributed systems.

Connection setup

When the device boots, it transmits `MSG_SETUP` messages on the serial line every 100ms. When the program starts, it waits for a valid `MSG_SETUP`, then replies with a `MSG_SETUP_ACK`. The system is then ready for use.

This design could lead to the serial input buffer on the host PC becoming full with `MSG_SETUP` messages, if the host is not started at the same time as the device. To avoid this, the host program flushes the input buffer on startup.

An alternative solution would be for the host to initiate communication. Such a design would also better reflect the underlying USB protocol, in which all communication is host-initiated. Having the host poll for updates may however increase latency.

Protocol diagram

A full diagram of the protocol can be found in section 3.7.

Name	Purpose
`INIT`	**Start state**. Open a serial connection to the device and wait for a valid `MSG_SETUP`.
`OPERATING`	Listen for, and process, `MSG_ENCRYPTED` messages representing keystrokes. Also handle `MSG_REKEY_INIT` by transitioning to state `REKEY`.
`REKEY`	Generate and display a password as described in subsection 3.4.4. Transitions immediately to `OPERATING`; an alternative implementation might perform some correspondence with the device to ensure the key agreement was successful.

Figure 3.4: List of states in the host.

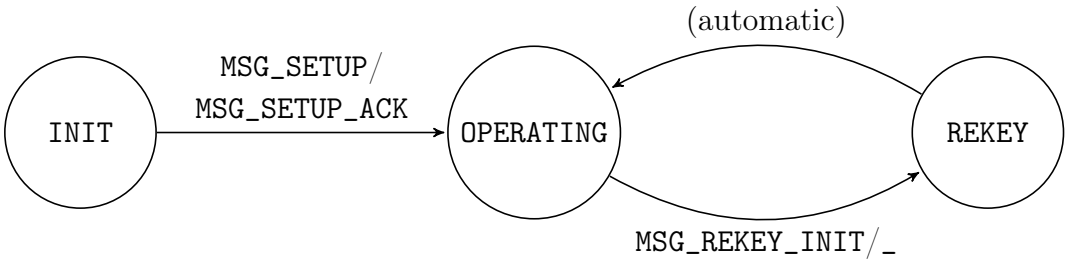

Figure 3.5: State diagram for the host. Edge labels indicate the input (if any) that triggers the transition, followed by the output (if any) caused by the transition.

Name	Purpose
`INIT`	**Start state**. Repeatedly send `MSG_SETUP` requests to the host and wait for a `MSG_SETUP_ACK` message.
`OPERATING`	Scan for keystrokes and send keystroke messages to the host. Also respond to the special keystroke entry sequence by sending `MSG_REKEY_INIT` and transitioning to state `REKEY`.
`REKEY`	Listen for and record keystrokes. When the Enter key is pressed, use entered keystrokes to generate new keys, then transition to `OPERATING`.

Figure 3.6: List of states in the device.

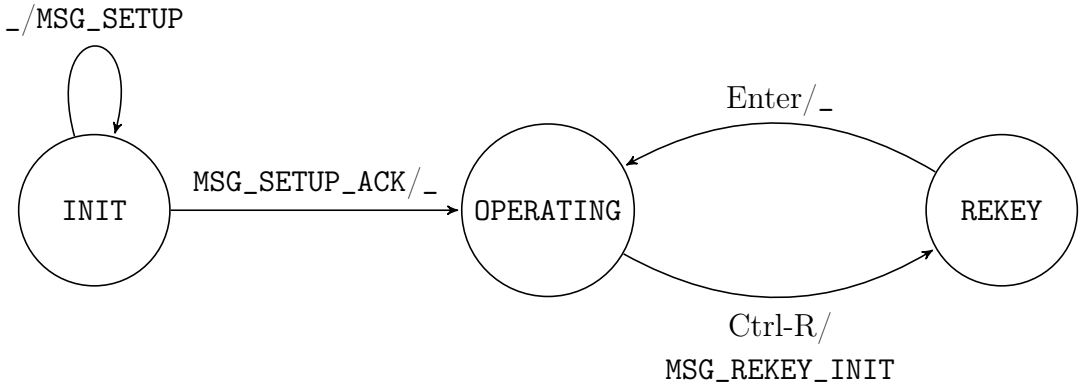

Figure 3.7: State diagram for the device. Edge labels indicate the input (if any) that triggers the transition, followed by the output (if any) caused by the transition.

3.3 Message security

In order to provide the secure channel, keystroke data is encrypted and authenticated.

The key cryptographic primitive used for both encryption and authentication is the *Advanced Encryption System* (AES). This is a *block cipher*, meaning that it takes a block of input data ('plaintext') of fixed size (128 bits) and a key (whose size depends on the variant of the algorithm) and deterministically produces a block of encrypted data ('ciphertext'), also 128 bits. The mapping of inputs to outputs is 1:1, and the transformation is easily reversible (the decryption operation), provided the key is known.

AES is a *symmetric* cipher, meaning that the same key is used to encrypt and decrypt data. This means both communicating parties must share a key; the way in which this is set up is described in section 3.4.

For this project I used AES-128, a variant in which keys are 128 bits long (alternatives are AES-192 and AES-256). All three variants are similar and the choice of a shorter key length variant was made to save memory on the microcontroller. Using a shorter key length theoretically makes the system more vulnerable to brute-force attacks, in which the attacker tries all possible keys, but as described in section 4.2, brute-forcing a 128-bit key is not feasible.

I decided to write my own implementation of AES-128 for this project. The reasons for this were two-fold: in order to learn about AES in detail, and to have an implementation which contains only the primitives needed for the project, minimising code size. This implementation was verified against the test vectors (sample input-output pairs) in the FIPS 197 standard [27], which specifies AES.[2]

In a system designed for production use, writing one's own version of cryptographic primitives like AES would be considered poor practice, as even a functionally correct implementation may be prone to *side-channel attacks*. For example, a poorly designed implementation might take varying amounts of time depending on the value of the key, thus leaking information to an attacker. In a production implementation, it would be preferable to use an implementation designed and vetted by experts, such as those in the NaCl library. [26] As AES is very widely used, some microcontrollers such as the Atmel XMEGA line even contain a hardware module for AES encryption and decryption. [6]

3.3.1 Message encryption

AES as described above is not by itself a suitable encryption solution, as it is only able to encrypt messages that are exactly the length of one block (128 bits). To encrypt anything else, the message must be padded to a multiple of the block length, split up into block-size chunks and then encrypted. There are various methods of doing this, known as *modes of operation*.

The simplest mode, *Electronic Codebook* (ECB), splits the message into 128-bit chunks and encrypts each chunk separately. The problem with this mode is that the same plaintext always encrypts to the same ciphertext. Using this mode to encrypt keystroke traffic might mean that every time a given key is pressed, the same ciphertext is generated. This makes the traffic vulnerable to *frequency analysis*: in this project, for instance, the most common encrypted message might represent the user entering an 'E', this being the most

[2] At this point I was somewhat hindered by the US government shutdown, which left the website of the National Institute of Standards and Technology offline. Fortunately I was eventually able to find a copy of the standard elsewhere.

common letter in English text. ECB mode also fails to obscure patterns in the data, as can be seen in Figure 3.8.

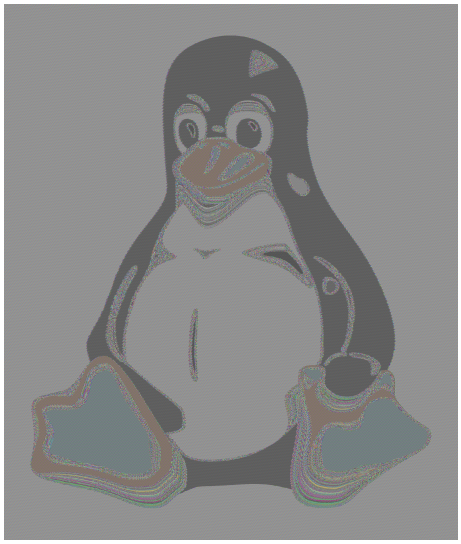

Figure 3.8: An image encrypted with a block cipher using ECB mode, showing the failure to obscure patterns in the data. Generated by Filippo Valsorda, based on an original by Wikipedia user Lunkwill. Licensed under CC-BY-SA.

There are many more sophisticated modes of operation which do not suffer from these problems. For this project I used *counter mode* (CTR). In this mode, the data is not directly encrypted with AES. Instead an incrementing counter is encrypted, generating a pseudorandom *keystream*. This keystream is combined with the data using XOR. To decrypt, the recipient generates the same keystream and applies XOR again, recovering the original data. Figure 3.9 shows this mode in use.

CTR mode has some advantages over other modes of operation. For this project, the most important is that it does not depend on the ciphertext or plaintext of block n to correctly decrypt block $n + 1$. This means that if a block of ciphertext is altered or corrupted in transit it will decrypt to nonsense, but subsequent blocks will decrypt correctly. Transmission errors therefore do not stop the system working, provided the counter is still incremented when a bad block is received.

This property proved useful for the project, as the USB serial line experiences occasional bit errors, and a single dropped keystroke is far less problematic than a complete failure.

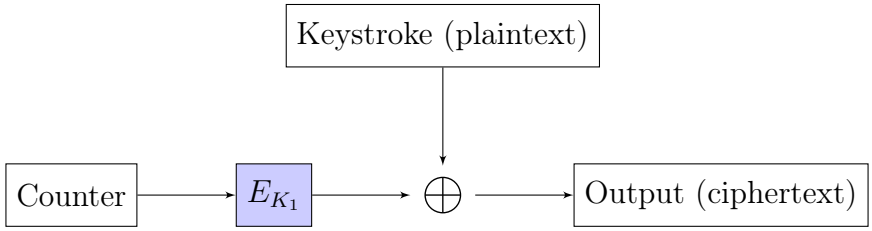

Figure 3.9: CTR mode encryption as used in this project. The symbol $\oplus$ denotes exclusive-OR (XOR).

If the error rate were higher, then it may be worth adding error correction, to supplement the error detection provided by the MAC.

All encrypted messages in this project are 16 bytes long; it is not a coincidence that this is the same as the AES block size. This choice simplifies encryption slightly, as the counter can be incremented exactly once for each message.

Securing CTR mode

A naive implementation of CTR mode might start the counter at zero whenever a connection is established. However, the same key and counter value should never be re-used together, as this results in multiple plaintext messages encrypted using the same keystream. If two plaintexts P_1 and P_2 are encrypted with the same keystream to give M_1 and M_2, then $M_1 \oplus M_2 = P_1 \oplus P_2$. This is a significant information leak, especially if the attacker knows the value of one of P_1 and P_2, in which case it is trivial to determine the other.

To prevent this occurring, either the counter must never be reset, or the key must be changed whenever a reset occurs. In my project I took the former approach: the counter is concatenated with a 32-bit *initialisation vector*, which is stored in non-volatile EEPROM[3] memory. The value of the counter is sent from the device to the host in the `MSG_SETUP` message when the connection is established.

If the 32-bit IV overflowed, messages could theoretically be encrypted with a counter value that has already been used, resulting in a vulnerability. Given the extra space in the `MSG_SETUP` message, the IV could indeed be larger than 32 bits. However, this is unnecessary as the IV is incremented only when the device boots, which is unlikely to occur anywhere close to 2^{32} times over the lifetime of the system.

Wear levelling

A more realistic concern is that each byte in the EEPROM is only rated to handle 10^5 writes, so the bytes storing the IV may wear out and fail after a few thousand startups. The solution to this would be to implement *wear levelling*, so successive writes are to different areas of the EEPROM. As the counter occupies only 4 bytes of the 1024 bytes available, there are plenty of places for it to be written, and this would prolong the life of the device considerably. Although I did not implement it, several libraries for wear levelling exist for the Arduino platform. [13]

[3] *Electronically Erasable Programmable Read-Only Memory*, a small amount of non-volatile memory which, unlike the flash memory used to store the program, can be written to by a program on the microcontroller.

3.3.2 Message authentication

As well as encrypting sensitive messages, it is necessary to authenticate them. This ensures that the encrypted message has not been altered in transit; messages which fail the authentication check on receipt are discarded. Authentication is important for several reasons:

- Block ciphers in counter mode are *malleable*[4]. This means that an attacker can affect the plaintext in a predictable manner by modifying the ciphertext. In counter mode, the ciphertext and plaintext are converted to one another by XOR with the keystream, so a bit flip in the ciphertext will cause a bit flip in the corresponding position in the plaintext. An attacker who knows the structure of the message could exploit this to make meaningful modifications. For example, flipping the bit that indicates whether one of the Shift keys is pressed would let an attacker alter the capitalisation of any letters typed.
- There have been attacks on cryptosystems which require an attacker to be able to make the system decrypt ciphertexts of their choice. While I have no reason to believe the encryption scheme used here is vulnerable to such *chosen-ciphertext* attacks, it is considered good practice to validate input before attempting to decrypt it. [15, p.112]

The algorithm used is a simple 'base case' of CBC-MAC. CBC-MAC works by encrypting the message with a block cipher in Cipher Block Chaining mode, in which the output of encrypting block n is combined with block $n+1$ using XOR. The last output block is used as the *Message Authentication Code* (MAC). For this application, however, the message to be authenticated (the encrypted keystroke message) fits within a single AES block, so calculating the MAC consists simply of encrypting it using one application of AES.

CBC-MAC is vulnerable if the messages over which the MAC is computed can vary in length. [15, p.127] In this situation, it is possible to forge a valid MAC for a message that was never sent, albeit subject to strict constraints. However, the messages in this protocol are fixed-length so this is not a concern.

The keys used for encryption and MAC computation are distinct. In some scenarios, using the same key can result in vulnerabilities. For example, if CBC mode is used for both encryption and MAC computation, the message may be malleable. In the design I use here, to the best of my knowledge there is no such vulnerability; separate keys are used as a precaution.

[4]This is a general property of *stream ciphers*, which generate a pseudorandom keystream and XOR this with the data. CTR mode can be thought of as a way of producing a stream cipher from a block cipher.

MAC limitations

The MAC prevents message forgery and modification; it does not prevent an attacker replaying, reordering or dropping messages.

It is common to include a *sequence number* in packets to detect out-of-order or repeated delivery. I chose not to include a sequence number explicitly, because the monotonically increasing counter used in CTR mode encryption provides this function already: if a message arrives out-of-order or is repeated, it will be decrypted with the wrong counter value, resulting (with overwhelming probability) in nonsense.

3.4 Key agreement scheme

The purpose of this scheme is to establish the two shared AES keys k_1 (for encryption) and k_2 (for authentication) on the device and the host.

As mentioned in subsection 2.5.1, the protocol is designed around the user transmitting a secret from the host to the device. When the user presses a special key sequence, the device prompts the host to generate a new secret. The host displays the secret on screen, for the user to type in.

Pressing the special key sequence also puts the device into 'secret entry mode' (state `REKEY` in Figure 3.6). In this mode, keystrokes are not sent to the host but are interpreted as the new secret to be used to determine k_1 and k_2.

3.4.1 Secure attention sequence

The key combination that triggers the secret entry mode must be one that is not commonly encountered in ordinary use. It should also ideally be reasonably simple and memorable. I chose right-control+R, the R standing for 're-key' or 'regenerate keys'. Right control specifically was chosen because it does not block other uses of the Ctrl+R key combination (for example to refresh a page in some web browsers), as it is still possible to use the left control key.

3.4.2 Initial implementation

In my initial implementation, the host generated the keys directly and displayed them in hexadecimal, and the user was required to type the entirety of both keys in on the keyboard. This was cumbersome and error-prone, as two 128-bit keys in hexadecimal is 64 keystrokes.

3.4.3 Password-based implementation

In this improved implementation, the host generates a random password of configurable length. This can be shorter than the two keys in the initial implementation, as it is 'stretched' using a *password-based key derivation function* (PBKDF). There are relatively few restrictions on the nature of the password: it can be any length from 1 to 128 characters (some upper limit is necessary due to the limited RAM on the microcontroller).

Password-based Key Derivation

A short password, even if randomly generated, will contain less entropy than a randomly-generated key. For example, while a 128-bit key has 2^{128} equally likely possible values, an 8-character password consisting of alphanumeric characters has $62^8 \approx 2^{47}$ values. Using such passwords could therefore leave the system vulnerable to a *brute-force attack* in which all possible values are tried. Brute-forcing 2^{128} values is impossible with current technology, but 2^{47} is feasible for a dedicated attacker, with 64-bit keys having been successfully brute-forced as early as 2002. [41]

To mitigate this threat, the keys are generated from the password by a *password-based key derivation function*, which is designed to run slowly, by performing many iterations of the underlying cryptographic operations[5]. The number of iterations is chosen so that a single key derivation is quick enough to be usable (a few seconds at most), but brute-force attacks are no longer feasible.

A threat that still remains is that a sufficiently well-resourced attacker could dedicate a large amount of computation to generate all the keys from a given range of possible passwords, meaning this effort can be re-used in multiple attacks.

To prevent this, a random and unique (but not necessarily secret) value called a *salt* is incorporated into the input of the PBKDF whenever it is used. Since the salt is changed for every use of the PBKDF, it is impossible to precompute a list of keys for a given range of passwords.

The salt is generated alongside the password, as described in subsection 3.4.4. It is sent to the device in a `MSG_SALT` message.

One disadvantage of a slow PBKDF for this project is that the difference in performance between the microcontroller in the device and a desktop processor is several orders of magnitude. In order for password entry to be usable, the PBKDF must run in no more than a few seconds on the device. The number of iterations must therefore be very low (testing indicated that around 2000 iterations can execute in 1 second). On the host, the PBKDF is much faster: approximately 500,000 iterations per second on an ordinary

[5]KDFs are related to the proof-of-work schemes used in cryptocurrency mining.

PC, using the same implementation.[6] A single key derivation using 2000 iterations could therefore be expected to take 0.0042 seconds.

Having so few iterations is suboptimal, as it makes brute-forcing easier. However, such an attack is still difficult: computing keys for all 2^{47} 8-character passwords would take $2^{47} \times 0.0042 \approx 5.9 \times 10^{11}$s, or approximately 19000 years. Longer passwords would exponentially increase this time.

The calculation above suggests that 8 character passwords are still difficult to brute-force. However, since the PBKDF relies on AES, it can be sped up significantly with hardware support. Many modern desktop processors support the AES-NI instruction set, which can provide encryption rates on the order of 10^9 bytes per second. [22] The bulk of the time involved in trying a given password is the 2000 AES iterations in the KDF, each of these being 16 bytes, so a back-of-the-envelope calculation gives $\frac{10^9}{2000 \times 16} = 31,250$ passwords per second. At this rate, enumerating all 8-character passwords would take $\frac{2^{47}}{31250} \approx 4.5 \times 10^9$s, around 142 years. While this is still beyond what is feasible on a single machine, an attacker with considerable resources (such as the money to rent hundreds or thousands of cloud server instances) could mount an attack in a more reasonable time. If this is a concern, it would be prudent to use longer passwords.

Some KDFs are deliberately designed to be difficult to accelerate in hardware. For example, `scrypt` [31] is designed to be memory-intensive, so providing special CPU support, or even building custom hardware, would not provide much of a speedup. Unfortunately the limited memory on the device (2KB) rules out use of algorithms like this.

Design of the password-based key derivation function

The PBKDF is a specialised form of *hash function*: a function which maps arbitrary-length inputs to fixed-length outputs.

Cryptographic hash functions have several security properties, not all of which are applicable for a KDF. The properties that are important for a KDF are:

- Being *one-way*, so compromise of one derived key does not lead to compromise of the input (and thus any other keys generated from the input). That is, it should be 'hard' to compute the input of the function given the output.[7]
- Being *entropy-preserving*: technically a function which always outputs a key of all zeroes is a valid KDF, but it is not very useful! Formally, the entropy of the output

[6] 1000000 iterations ran in a mean of 2.109s with standard deviation 0.035s, $n = 10$. Tests run on an Intel Core m3-6Y30 CPU on Ubuntu 18.04, code compiled with GCC at optimisation level O2.

[7] Ideally one would like the function to be a *one-way function*, computable in polynomial time but with an inverse provably only computable in superpolynomial time. Unfortunately the existence of these functions is an open problem, and their existence would imply P $\neq$ NP.

should be approximately equal to the lesser of the entropy of the input and the length of the output.[8]

For this algorithm, the underlying cryptographic primitive is the AES block cipher, as used for message encryption and authentication. AES was chosen because it is already present in the program, and minimising the size of the program is important if it is to fit into the limited program memory on the microcontroller.

AES is used in a *Davies-Meyer construction* [19, p.110] to construct a *compression function*. This is a function which transforms two inputs of length n to a single output, also of length n. Note that despite the name, such functions are not to be confused with algorithms for data compression, as they are not designed to be easily reversible or to preserve information about the input.

Given a block cipher encryption function $E : \{0,1\}^l \times \{0,1\}^n \to \{0,1\}^n$ with key size l and block size n, the Davies-Meyer construction defines the compression function $C(K, M)$ as $E_K(M) \oplus M$. For AES-128 as used in this project, conveniently $l = n$, but this is not a requirement for the construction to work. The operation of the Davies-Meyer construction can be seen in lines 6 to 10 of Algorithm 1.

In order to construct the desired *variable-length hash function*, the compression function is used in a modified *Merkle-Damgård construction*. [29, p.91] In this construction, the input is padded to a length divisible by n and split into blocks of length n.[9]

The blocks are then combined one at a time, using the Davies-Meyer construction, with the salt as the initial value.

Finally, the output of the Davies-Meyer compression is used to encrypt the constants `0x00` and `0x01`, giving the keys k_1 and k_2.

The details of the PBKDF are shown in Algorithm 1. The cost factor I controls the number of iterations of AES applied in the compression stage (lines 6-10), and thus the running time of the algorithm.

[8] In other words, the output should not have lower entropy than the input, unless constrained by the length of the output (an n-bit value cannot have more than n bits of entropy).

[9] In a standard Merkle-Damgård implementation, the length of the input is then appended as an extra block, so that no input to the next stage is a suffix of another. This strengthens the proof of collision resistance. [29, p.93] However, collision resistance is not a significant concern for a KDF, so this step is skipped.

Algorithm 1: Davies-Meyer based KDF

Input: Password P, Salt S, Cost factor I
Output: 128-bit keys k_1, k_2

$n \leftarrow \lceil |P|/128 \rceil$
$P \leftarrow P$ padded with $128n - |P|$ zeroes
for $i \leftarrow 1$ **to** n **do**
    $P_i \leftarrow i^{\text{th}}$ 128-bit section of P
$C_0 \leftarrow S$
for $i \leftarrow 1$ **to** n **do**
    $T \leftarrow C_{i-1}$
    for $j \leftarrow 1$ **to** I **do**
        $T \leftarrow \text{AES}_{P_i}(T)$
    $C_i = T \oplus C_{i-1}$
$k_1 \leftarrow \text{AES}_{C_n}(0)$
$k_2 \leftarrow \text{AES}_{C_n}(1)$

3.4.4 Random password generation

The protocol described above relies on being able to generate unpredictable random values, for keys (in the original design) or the password and salt (in the later design).

Such values are generated using a *cryptographically secure pseudorandom number generator*. Informally, this is an algorithm which generates a sequence of values that are indistinguishable from a comparable truly random stream, such as that produced by tossing a coin repeatedly. More formally, a CSPRNG should satisfy the *next-bit test*: given the first k bits of an output sequence, there is no polynomial-time algorithm that can predict the next bit with probability greater than $0.5 + negl(l)$. *negl* is a negligible function which, as l tends to infinity, tends to zero faster than $1/poly(l)$ for any polynomial *poly*.

The CSPRNG used is that provided by the Linux kernel and made accessible via the `getrandom` system call[10]. In recent versions of the kernel this is based on the ChaCha20 stream cipher. The entropy used to seed the CSPRNG (i.e. the source of 'true' randomness for what is ultimately a deterministic algorithm) is gathered from device drivers, ultimately coming from the timing of unpredictable events such as network packet arrival.

Generating high-quality random values suitable for cryptographic use on a microcontroller is more difficult than on a PC, because of the relative lack of entropy. Some sources even recommend connecting an external 'noise-generating' circuit to provide sufficient entropy.[11] [37]. For this reason, the protocol is designed so that all random number

[10] The same generator is accessible via (and commonly known as) the special file `/dev/random`.

[11] Some specialised microcontrollers, such as the Atmel secureAVR series, include a built-in noise source, or even a full hardware random number generator.

generation takes place on the host.

From bytes to passwords

The output of `getrandom` is an arbitrary quantity of random bytes. To convert this to a password, each 4-byte block is interpreted as an unsigned 32-bit integer and taken modulo 62. The result is used as an index into a 62-element array containing the ASCII characters A-Z, a-z and 0-9.

3.5 Simulating keystrokes with `uinput`

Once a keystroke event has been received, authenticated and decrypted, it is time to act on it by injecting a keystroke on the host system. For this I use the `uinput` Linux kernel module [42], which provides an API for creating and controlling virtual input devices from userspace (i.e. outside the kernel).

3.5.1 Configuring the virtual keyboard

Before `uinput` can be used to inject keystrokes, it is necessary to configure the virtual input device. To the rest of the system, this appears identical to the user plugging in a real keyboard.

Configuration is performed by a series of `ioctl` calls. These are system calls that allow arbitrary communication between userland programs and kernel modules such as device drivers. The first argument of an `ioctl` is the file descriptor of the device being manipulated (following the UNIX philosophy of devices as files); for `uinput` this is the special file `/dev/uinput`. Once configured, keystrokes are simulated by writing to the same file using the `write` system call.

Device IDs

As part of the configuration process, it is necessary to specify the vendor and device IDs, and the human-readable device name. Vendor IDs are allocated to hardware manfacturers by industry bodies such as the USB Implementers' Forum (USB-IF), to ensure all types of device have a unique identifier. Operating Systems frequently use these IDs to choose the correct driver for a device.

Such IDs are not cheap: USB-IF charges at least $3,500 for a Vendor ID. [12] Unsurprisingly, I did not apply for an official Vendor ID for this product. Instead I used one

which is reserved for development purposes by Objective Development, a company which produces USB libraries. [11] As this ID is shared, there is no guarantee it will not conflict with other devices, but it will at least not conflict with devices produced by licensed manufacturers who have their own Vendor ID.

3.5.2 Keycode conversion

One of the tasks performed by the host software is to translate keystroke information from the format used by the USB HID specification to that used by the Linux kernel and `uinput`.

The keyboard events received from the device encode the keystroke data as two 1-byte values: the *scancode*, which corresponds to the key pressed or released, and the *modifiers*, in which each bit indicates whether a given *modifier key* (shift, control, etc.) was pressed at the time of the event. For example, pressing Ctrl-C would generate a single key-down event, with the scancode corresponding to the key 'C' and the modifiers indicating that the (left or right) control key was down. Presses of modifier keys do not generate keystroke events by themselves.

This format does not correspond with that required by `uinput`. In order to convert the keystroke to a format suitable for `uinput`, it is necessary to do the following:

- Convert the scancode to the corresponding Linux input event code. This is a straightforward lookup in an array indexed by scancode.
- Compare the modifier state to that in the previous keystroke (for the first keystroke after startup, the modifier state is assumed to be `0x00`, i.e. no modifiers pressed). If it has changed, insert key-down or key-up events for the relevant control key(s), before emitting the event for the non-control key.

For example, if the host receives a key-down message with scancode `0x04` and modifiers `0x01`, this corresponds to a press of the key 'C' with the left control key down. When receiving this, the host converts the scancode to an event code, in this case `0x1E`, then compares the modifier state to that at the time of the previous keystroke. Assuming the previous modifier state is `0x00` (i.e. no modifier keys were pressed), the host determines that the bit corresponding to the left control key has changed from 0 to 1. It therefore emits a key-down event for the left control key, followed immediately by a key-down event for the key 'C'.

3.6 Repository Overview

The repository consists of two programs, both written in C. The `device` program runs on the microcontroller and the `host` program runs on the host PC.

Both programs are structured for use with the CMake build system, which is integrated into CLion (a C and C++ IDE similar to IntelliJ IDEA).

Neither program is based on an existing project; both are written from scratch. However, several existing libraries are used.

3.6.1 Device program

The device program is implemented as a state machine, in `src_device.ino`.

The device program is written mainly in C, with a small amount of C++ to interact with the USB Host Shield library, which is written in C++.[12] The CMake build system is configured to cross-compile to the AVR architecture (used by the microcontroller) using the GCC compiler. Loading the compiled program onto the microcontroller over USB using `avrdude` is also performed automatically as a step in the build process.

Developing for AVR microcontrollers is very similar to developing for an ordinary x86 Linux system. The `avr-libc` project [1] provides a large subset of the standard GNU `libc`.

The following libraries are used:

- `avr-libc` [1]: provides a subset of the standard C library. Licensed under a custom permissive license based on the Berkeley license. [40]
- USB Host Shield library [36], licensed under the GNU General Public License (GPL).[13]

3.6.2 Host program

The host program, like the device program, is at its core a state machine. The state machine, and the entry point of the program, are in `host.c`.

[12] Strictly speaking, the entire device program is compiled as C++; the C features used are a subset of C++, although this is not a property of all C programs.

[13] The use of a GPL library requires the entire project to be released under the terms of the GPL. This can be problematic in commercial applications if the manufacturer wishes to keep their source code secret, so for a production implementation it may be worth using a different library.

The following libraries and code fragments are used:

- `arduino-serial-lib` [21]: a small library which configures the serial port for communication with the Arduino board. Licensed under the MIT License.
- The `uinput_setup` and `uinput_emit_event` methods in `uinput.c` are based on example code [42] from the Linux kernel documentation. Licensed under the GPL v2, as part of the Linux kernel source.

Some code is common to both the host and device programs, such as the AES implementation in `aes.c`/`aes.ino`. Originally I hoped to keep a single copy of this code and include it in both projects; this did not work well with the CMake build system, so it was necessary to duplicate it. The duplicated code was excluded when calculating the line count in the Proforma.

3.7 Protocol diagram

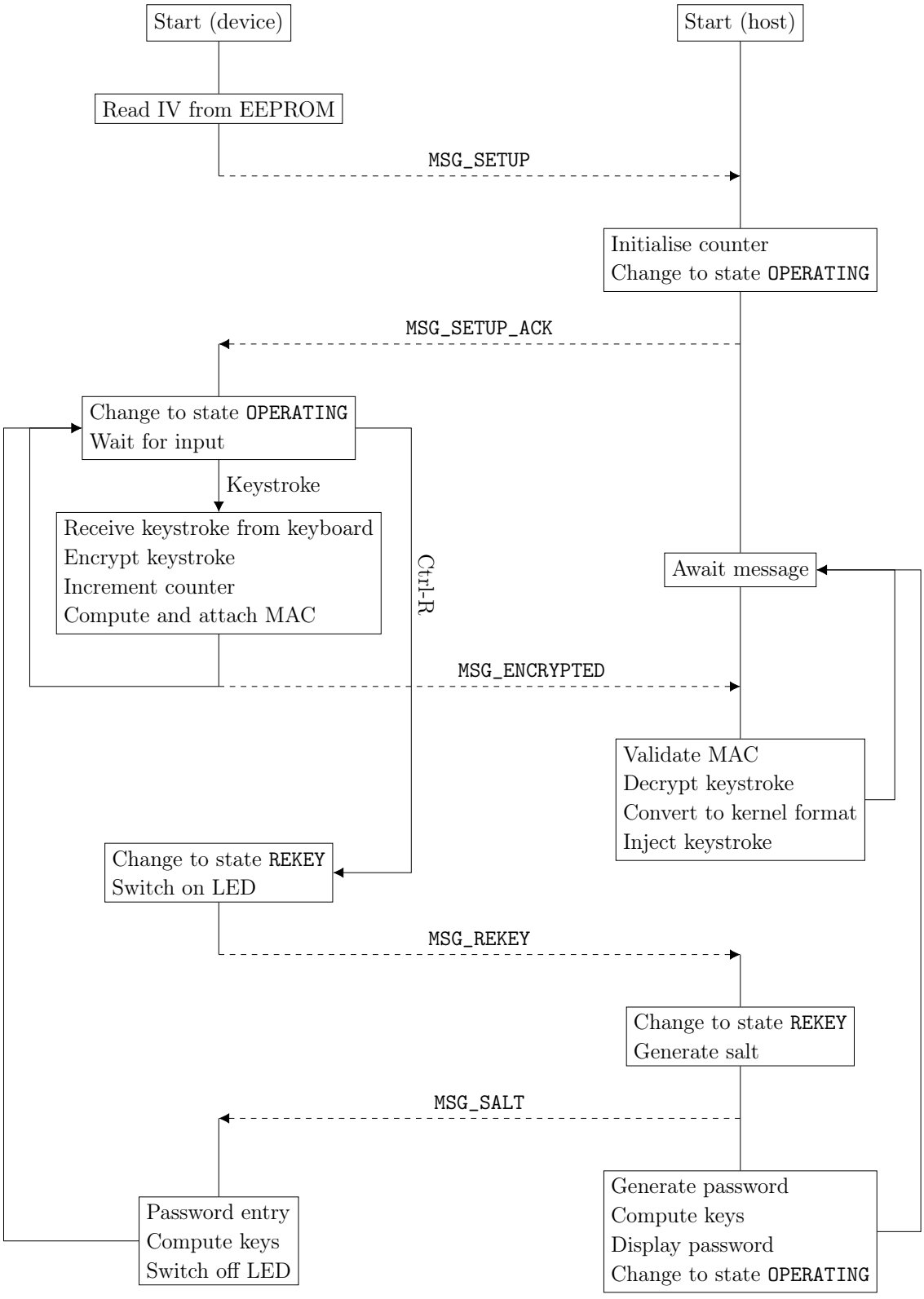

Figure 3.10: Overview of the programs on the device and host, and the messages exchanged between them.

Chapter 4

Evaluation

4.1 Timing

4.1.1 Motivation

One of the requirements for the system is that it can be used like an ordinary keyboard. For this to be true, the latency of the system (the time from a key being pressed to the PC responding to the keypress) must be comparable to that of an ordinary keyboard. A timing analysis was carried out to determine the additional latency of the system compared to a keyboard directly attached to a PC.

4.1.2 Methodology

To measure the latency, I used a high-framerate camera, specifically the camera on an iPhone X, which can capture video at 240 frames per second.

In order to indicate when the keystroke is registered (and timing should be stopped), I added another LED to the device, and another message (`MSG_KEYSTROKE_ACK`), from the host to the device, to instruct the device to blink this LED. The camera is used to measure the time between the key being pressed and the LED illuminating.

After filming a number of keystrokes, I stepped through the videos frame-by-frame to determine the number of frames between the key press (measured at the point where the key reaches the bottom of travel) and the LED illuminating. An example frame can be seen in Figure 4.1. As the frame rate is known and constant, the number of frames is easily converted to a latency value.

Part of the aim of this analysis was to compare the system against a keyboard connected directly to a PC. For this, a subset of the normal host program was run, which simply

Figure 4.1: An example frame from the timing analysis showing the indicator LED (yellow) illuminated. The number at the bottom is the frame number.

listens for a keystroke (using the `getchar` function from the C standard library) and sends a `MSG_KEYSTROKE_ACK` to the Arduino. The Arduino was programmed with a simple program to listen for this message and blink the LED (`bare-keyboard-timing.ino`).

An alternative to blinking the LED would be to simply wait for the character to appear on the PC display. However, the refresh rate of my display is only 60Hz, and it is very rare to see displays with refresh rates above 120Hz. The LED is therefore necessary so as not to lose precision.

4.1.3 Measuring jitter

The tests were run on an ordinary Ubuntu Linux installation with no other applications open. However, as this is not a real-time OS, process scheduling is unpredictable and some jitter (variance in latency) is to be expected.

Another source of jitter comes from the fact that USB is host-driven, and the host *polls* the device at a regular interval for new events (8ms is common). For this device, there are in fact two stages of polling: the Host Shield polls the keyboard, then the host polls the microcontroller.

Jitter does not represent a flaw in the testing methodology: the aim of the experiment is to measure performance in a realistic situation, and this jitter is representative of normal use of the system.

4.1.4 Limitations

This experiment was designed to compare the latency of the system with that of an ordinary keyboard, not to determine the latency in absolute terms. The values measured are likely to all be overestimates of the true latency by a constant factor, due to the time taken for the `MSG_KEYSTROKE_ACK` to be transmitted to, and processed by, the microcontroller.

Measuring the time of the keystroke is a potential source of inaccuracy. At 240fps, a keystroke takes many frames, so I chose to start counting from the point where the key reaches its lowest position. The keystroke is actually registered by the keyboard when the key is approximately 1mm above the lowest position, so the timing starts a frame or two late. Measuring the exact position of the true starting point, and determining when the key has passed it in each video, would be extremely difficult.

If the key press speed is the same in every trial, then this will only result in a constant offset from the true latency, which is acceptable as the aim is comparison. I made every effort to keep the key press speed the same, but this cannot be guaranteed.

4.1.5 Results

When the keyboard was attached via the trusted-path system, the mean latency was 90.2ms, with a standard deviation of 8.7ms. ($n = 20$).

When the keyboard was plugged into the PC directly, the mean latency was 48.5ms, with a standard deviation of 8.6ms ($n = 20$).

These results suggest that the system does add a small amount of latency (approximately 40ms).

Whether 40ms of additional latency is theoretically perceptible is the subject of controversy, with figures from 13ms to 100ms being quoted as the minimum latency that is noticeable in a user interface. [10] Anecdotally, however, I could not perceive the extra latency.

The standard deviation for both sets of measurements is very similar. This suggests that the trusted-path system does not introduce additional jitter.

Raw data can be found in Appendix B.

4.2 Security analysis

In this section I present a qualitative security analysis of the system.

The aim of this section is to systematically consider possible attack vectors which, given the threat model in section 2.4, could be used to 'break' the system. Although this effort aims to be as systematic as possible, it is impossible to guarantee that nothing has been overlooked: quantitatively 'measuring' security is not a solved problem.

What constitutes a 'break' is based on the requirements in section 2.3. The definition used for the analysis is that an attacker is able to 'break' the security of the system by doing any of the following:

- Determine which keys are pressed on the keyboard with a success rate beyond that of guessing. This does not have to take place in real time as the keys are pressed.
- Cause a false keystroke to be registered on the host system, either altering a legitimate keystroke or injecting a new keystroke entirely.
- Replay the entry of any sequence of one or more keystrokes previously entered on the keyboard.

In order to enumerate possible threats in a reasonably systematic fashion, I use the concept of *Attack Trees* [32], devised by cryptographer Bruce Schneier. In an Attack Tree, the overarching goal of the attacker (e.g. 'steal money from the bank') is represented by the root node. Ways of achieving this goal (e.g. 'take money from the safe') are represented by its child nodes, and leaves represent individually actionable tasks (e.g 'bribe bank manager for safe combination').

The directly actionable tasks (leaf nodes) on the tree are detailed and evaluated after the tree (Figure 4.2).

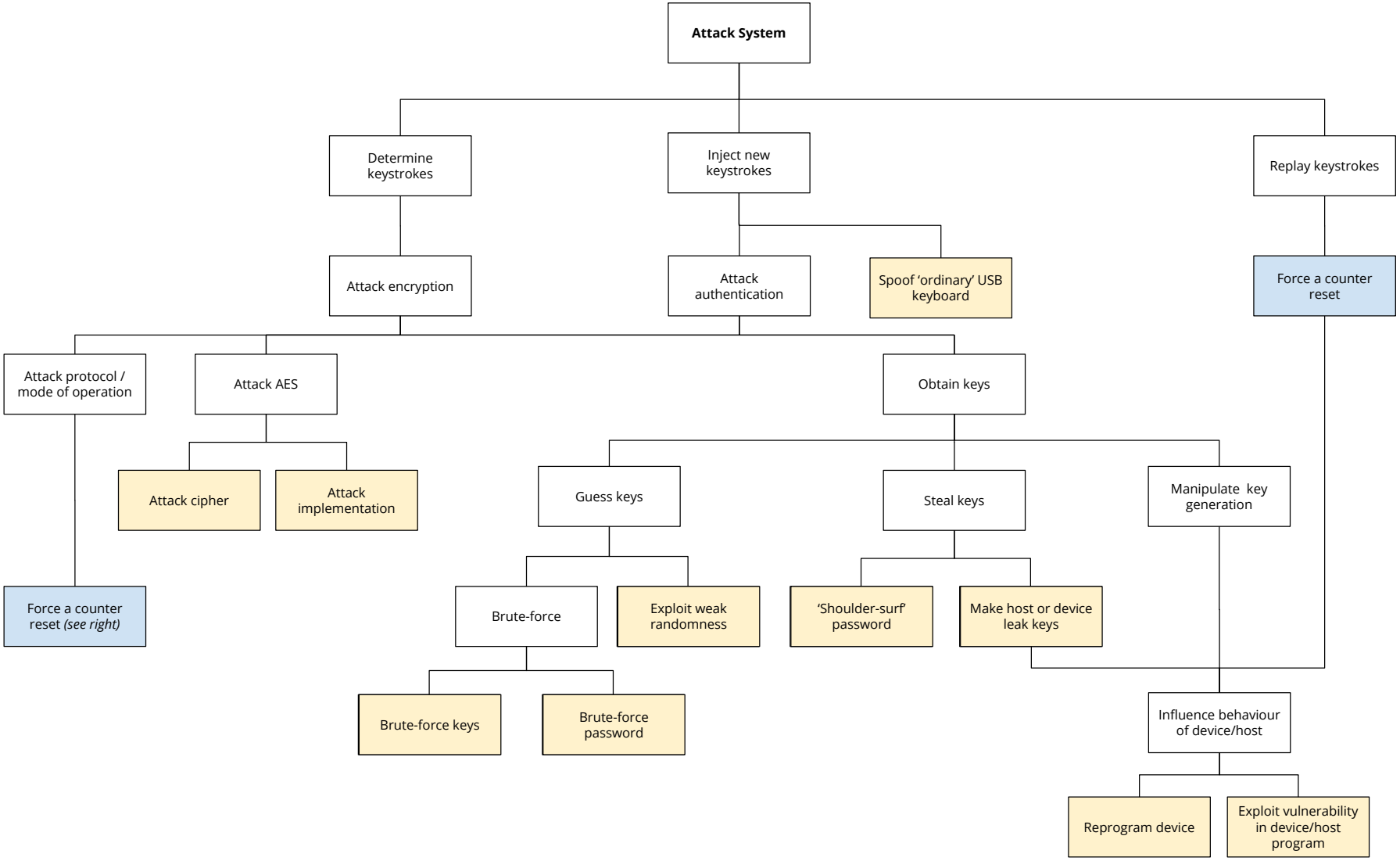

Figure 4.2: Attack Tree for the system. Yellow nodes indicate directly actionable tasks. The two blue nodes represent the same attack, and are duplicated to avoid cluttering the tree.

1. **Attack AES implementation**

   As mentioned in subsection 3.3.1, I implemented AES myself as a learning experience. Whilst my implementation is functionally correct, it may contain *side-channel vulnerabilities* which could leak information about the plaintext and/or the key.

   The most common such attack is known as a *timing attack*, and exploits variation in the execution time of the algorithm with different inputs.

   Timing attacks against even well-established AES implementations have been developed; a 2005 attack by cryptographer Daniel Bernstein exploits timing variations caused by CPU caching to determine a key by timing the encryption of chosen plaintexts, using the AES implementation in the popular OpenSSL library. [7]

   Since the aforementioned attack, considerable effort has however gone into developing constant-time cryptography. The AES implementation in the NaCl cryptography library, for example, operates without data-dependent branching or data-dependent array lookups [26], and as such is resistant to attacks of the form described above. A production implementation of this product would probably use a library like this.

   A related family of side-channel attacks is based on *power analysis*. This involves measuring the power consumed by a piece of hardware as it performs cryptographic operations.

   Power-based attacks are compatible with the threat model in section 2.4. The device is powered over the USB connection, so an eavesdropper could measure the current draw with an inline ammeter.

   Power analysis is possible, but requires a large amount of operations. For example, [23] describes the recovery of a 128-bit AES key from an implementation of a very limited part of an AES encryption (a single `AddRoundKey` followed by a single `SubBytes`), taking 25 minutes and using known, but not arbitrarily chosen, plaintexts. Ultimately this relies on the same vulnerabilities exploited in timing attacks; that control flow (and thus power consumption) can, in naive implementations, depend on secret values.

   An alternative, physical, mitigation would be to power the device from a separate power supply, assuming this could not also be compromised.

2. **Attack AES cipher itself**

   This attack would exploit vulnerabilities in the AES algorithm. At the time of writing, there are no known vulnerabilities of practical interest, so this attack is not considered feasible.

   While there are no known practical vulnerabilities, cryptographers have discovered weaknesses which are of academic interest, i.e. potential attacks requiring less effort than brute-force. For example, Bogdanov et. al. [8] discovered a theoretical key-recovery attack that requires only $2^{126.1}$ operations, instead of the 2^{128} to brute-force an AES-128 key.

It is plausible that weaknesses like these may one day be developed into practical attacks. Given how widely AES is used, if such a practical attack were developed, there would most likely be more lucrative targets than this system.

3. **'Shoulder-surf' password**

   'Shoulder-surfing' attacks involve observing data on the target's screen, or the target's keystrokes. These are commonly used to obtain bank PINs from ATM users. The attacker may be literally looking over the victim's shoulder, or a concealed camera may be used.

   This attack is difficult to mitigate in the current design of the system. However, an alternative design could be used, in which the user transfers data that does not need to be secret. This is discussed further in subsection 2.5.1.

4. **Brute-force AES key(s)**

   A *brute-force* attack, as mentioned in section 3.3, involves trying all possible values for a key until the correct one is found. This is feasible for short keys: the COPACOBANA hardware, for instance, can successfully brute-force a 56-bit key (used by the DES cipher) in under 9 days for a cost of less than €10000, [20] and given sufficient computing power, 64-bit keys are vulnerable. [41].

   A brute-force attack on 128-bit keys is theoretically possible, but considerably beyond anything that has been accomplished today. Increasing the key length to 256 bits would make a brute-force attack thermodynamically impossible until computers are 'built from something other than matter'. [33]

5. **Brute-force password**

   As described in subsection 3.4.4, the password is randomly generated with a configurable length, and an 8-character alphanumeric password has $62^8 \approx 2^{47}$ possible values, which it is feasible for a well-resourced attacker to try.

   To mitigate this threat the PBKDF is designed to run slowly. A single key derivation operation, as needed for legitimate use of the system, does not take an unduly long time (at most a few seconds), but brute-forcing keys for a large set of possible passwords is no longer feasible.

   As described in subsection 3.4.3, deriving keys from all possible 8-character passwords would take many years on an ordinary PC, but is within the realm of possibility for a well-funded attacker. Longer passwords would render this attack impossible for even these attackers.

6. **Exploit random number generation weaknesses.**

   The claims regarding the difficulty of brute-forcing the password rely on the assumption that it is randomly generated in a way that cannot feasibly be predicted.

   The password is generated using the CSPRNG built into the Linux kernel, accessed via the `getrandom` system call. This is considered adequate for the generation of cryptographic secrets, provided sufficient entropy is available to initialise (seed) it.

The security of the random password generation is discussed in detail in subsection 3.4.4.

Given how widely the Linux kernel CSPRNG is used for key generation, a major vulnerability would result in a similar situation to what would result from a break in AES: it may compromise this system, but there would be plenty of higher-value targets.

7. **Reprogram device to leak data**

   The ATMega328P microcontroller used on the Arduino platform is shippped pre-programmed with a *bootloader* program which, in combination with the on-board USB hardware, allows the microcontroller to be reprogrammed over the same USB connection with which it connects to the host. [5]

   An attacker with the ability to transmit on the USB connection could use this facility to replace the program running on the microcontroller. An obvious replacement program would be one that appears to operate identically to the standard program but can also communicate with the attacker to leak data.

   This is a major vulnerability, but fortunately it is easily fixed by using a microcontroller without the bootloader program. Once this is done, changes to the code are only possible with programming hardware. Such an attack would involve physically tampering with the device, and thus falls outside the threat model.

   It is possible to remove the bootloader on an Arduino, using the aforementioned programming hardware, to give a 'bare' microcontroller. I elected not to do this because the ability to reprogram over USB is very useful for development.

8. **Exploit vulnerabilities in host program**

   The host program necessarily accepts and processes input from the device, so an attacker could send inputs that are crafted to cause the host program to malfunction in some way.

   Common vulnerabilities include *buffer overflow* attacks, in which the receiving program is tricked into writing data beyond the end of a fixed-size input buffer. The oversized input may overwrite other data on the stack (attacks on heap-allocated buffers are also possible but somewhat more complex), including most importantly the return address, causing execution to jump to an attacker-chosen address. This can lead to near-arbitrary code execution. Buffer overflows can be avoided if all reads of untrusted data are of fixed length, which is the case in this program.

   In order to minimise the damage caused if the program is compromised, it is good practice to run it with the minimum possible privileges. Doing so ensures that even if an attacker is able to execute arbitrary code via the exploited program, their access to the rest of the system is limited.

9. **Spoof a directly-attached keyboard**

   An attack vector that allows arbitrary keystroke injection is for the attacker to present itself to the host PC as a USB keyboard, instead of the USB serial device that

the system normally uses. A variation on this would be to pretend to be a USB hub, with the trusted path keyboard as one connected device, and an ordinary keyboard as another. Once connected as a keyboard, the attacker can inject keystrokes by sending USB Human Interface Device (HID) messages, bypassing the host software entirely.

As well as entering attacker-chosen data, malicious keystroke injection can lead to arbitrary code execution, as the malicious device can enter keystrokes that cause the system to, for example, open a terminal and download and run an arbitrary program from the Internet. This attack vector is widely known; programmable keystroke-injection devices disguised as USB flash drives are available for under $50. [14]

A straightforward mitigation is to disallow USB HID devices such as keyboards from connecting to the host PC. On most Linux systems this can be achieved by configuring the `udev` device manager. Doing so might be inconvenient, but restrictions on USB devices are already common in high-security corporate environments. [39]

A handful of other nodes represent attacks that are not directly actionable but also bear considering.

10. **Attack encryption / authentication protocol, or mode of operation**

    The counter (CTR) mode of AES operation as described in subsection 3.3.1 is vulnerable if used incorrectly. Specifically, if two or more streams of plaintext are encrypted with the same key and initialisation vector (IV), then the XOR of the ciphertexts will be identical to the XOR of the plaintexts. If one of the plaintext messages is known, the other can then be recovered.

    To exploit this attack, the attacker would need to force an IV or counter reset. This is investigated under the entry for 'Force an IV counter reset' (item 11).

    There may be other vulnerabilities in the protocol. Given the protocol's simplicity it is unlikely that there are any major issues, but with a qualitative analysis this cannot be guaranteed. To some extent, it would be possible to use formal methods to prove the absence of vulnerabilities; this is discussed further in subsection 4.2.1

11. **Force an IV or counter reset**

    In order to perform some attacks, the attacker must be able to alter the value of the counter used to encrypt. This could involve altering the initialisation vector, which under normal operation is monotonically increasing and stored in non-volatile EEPROM memory, or the counter, which is stored in RAM and reset to zero every time the device starts up. By design, the protocol does not include any way to reset these counters, but it would be possible to use the same technique described in 'Reprogram device to leak data' to write a new value to the EEPROM, or indeed to reprogram the device with a new program that resets the counter under certain conditions.

The attacker can tamper with the IV value when it is sent from the device to the host in the `MSG_SETUP`. However, the host only uses the counter for decryption, so tampering with it would simply cause any messages received to be incorrectly decrypted.

If encrypted messages were to be sent in both directions, as might be the case in a more complex protocol, this single counter would be vulnerable, as the host would encrypt using a counter value that may have been tampered with by an attacker. It would be necessary to maintain two counters, one for each direction of communication.

12. **Trick user into entering password outside designated password-entry mode**

    In password-entry mode, any keystrokes entered on the keyboard are interpreted as a password for generating a new set of keys, and accordingly are not sent to the host in keystroke messages. If an attacker could trick the user into entering a password when the device is not in password-entry mode, it would be sent, encrypted, to the host.

    In normal operation the password should never be sent to the host, but doing so (over the encrypted link) is only dangerous in two scenarios:

    - If the existing encryption key is known, the attacker can determine the password. If the user subsequently enters the same password in password-entry mode and it is used to derive new keys, the attacker will also be able to derive the same new keys and the system will remain compromised.

      This scenario requires the encryption key k_1 (although not necessarily the authentication key k_2) to be compromised already, as well as a means of tricking the user into entering the password at the wrong time. Thus it is difficult to exploit for relatively little gain.

    - If the host is running a malicious program which logs keystrokes entered, then this will log the password, from which the key can be derived.

      This scenario requires the host to be compromised already, and does not provide a significant advantage to the attacker beyond what would already be available from a compromised host, although it may allow the decryption of past keystrokes.

    To reduce the likelihood of either of these scenarios occurring, the device is fitted with a green LED which illuminates only when in password-entry mode. Ensuring that the password is only entered when this LED is illuminated is, however, an issue of user compliance, which is difficult to guarantee.

4.2.1 Summary

The security analysis identified some issues with the implementation of the device as-is. In particular, the ability to either reprogram the microcontroller or spoof an 'ordinary' keyboard are very concerning. Fortunately, both these issues are easy to mitigate. The analysis did not identify any attacks which are both feasible to execute and difficult or impossible to mitigate.

Assuming the fixes described are applied, the system appears to meet the security requirements in section 2.3.

While the Threat Tree methodology aims to be as exhaustive as possible, it is ultimately a qualitative method which is not guaranteed to find every possible vulnerability. It remains possible that there are feasible attacks not enumerated above.

An interesting extension to this project would be to evaluate parts of the system using formal methods. Techniques such as BAN logic express protocols in a form which can be reasoned about mathematically, the aim being to prove security properties and expose weaknesses. [9] This is the closest thing to an exhaustive security analysis technique.

While these formal methods are useful, they are not a panacea. [2, p.90] Their scope is usually limited to the communications protocol and potentially the underlying cryptography; as such they would not expose threats related to the implementation. Furthermore, even formal proofs of security can contain errors, or rely on assumptions which later turn out to be false: 'if it's provably secure, it probably isn't'. [2, p.857]

4.3 Usability analysis

This section aims to briefly compare the system against the usability requirements in section 2.3.

U1 *The system must provide a keyboard-based input channel which is usable as an ordinary keyboard. After a setup phase, the user should be able to forget they are using a specialised input device, i.e. the device should be transparent.*

This requirement is mostly met; it is perfectly possible to use the system like a normal keyboard. There are a few minor issues:

- The secure attention sequence (right-Ctrl+R) used to start the key agreement means that it is not possible to use the system to type this exact key sequence. It is, however, possible to use the left Ctrl key instead for almost all uses.
- The status LEDs on the keyboard (Caps Lock, Num Lock etc.) do not currently work. I spent several days on this issue, but the USB Host Shield library is

very poorly documented and I decided continuing was not an effective use of my time.

U2 *The setup procedure, if it requires user input, should consist of simple and easy-to-follow instructions.*

This requirement is met. The user is only required to type in a short password and press Enter. Instructions for doing so are printed to the console.

Some users might find a graphical user interface less intimidating than the command line, so it would make sense to add one in a production implementation. However, this project was intended as a proof of concept, and a polished UI was not an aim.

Chapter 5

Conclusion

Overall, this project has met its success criteria, as described in the project proposal (Appendix C). I have constructed a system which provides a keyboard-based input channel to a Linux PC and is secure against keyloggers. I have also measured the timing characteristics of the system compared to an ordinary keyboard, and determined that system adds a small but acceptable amount of latency and does not significantly increase jitter.

The system is secure against most of the issues raised in the analysis. Some vulnerabilities are present due to the nature of the prototype, but would be easy to remedy in a commercial product.

It would be too bold to claim that the system is 'secure' in an absolute sense. While some aspects could be formally verified (as discussed in subsection 4.2.1), there is no way to guarantee the security of a system this complex.

It also bears remembering that the threat model against which the system was evaluated was quite narrow, and a realistic attacker may not be limited to planting keyloggers. Nonetheless, the system provides a valuable security benefit, at low cost.

5.1 Potential improvements

Implementing the host program as an ordinary userspace application made for relatively simple development, but meant that the system cannot protect login passwords. An improvement here would be to implement the host program in a way that meant it could run earlier in the boot process, for example as a driver in the Linux kernel.

The use of a password-based key derivation algorithm is not ideal, as the password has to be kept secret. Alternatives such as Password Authenticated Key Exchange (discussed in subsection 2.5.1) use the password for verification, and it does not have to be kept secret. This would make the system more resistant to 'shoulder-surfing' attacks.

Bibliography

[1] *AVR Libc Home Page*. https://www.nongnu.org/avr-libc/.

[2] R. J. Anderson. *Security Engineering*. 2nd. 2001.

[3] *Arduino USB Host Shield*. https://store.arduino.cc/arduino-usb-host-shield.

[4] *Arduino Uno Rev3*. https://store.arduino.cc/arduino-uno-rev3.

[5] *Arduino as ISP and Arduino Bootloaders*. https://www.arduino.cc/en/tutorial/arduinoISP.

[6] Atmel Corporation. *Atmel AVR XMEGA A1 ATxmega64A1 and ATxmega128A1 Device Datasheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8067-8-and-16-bit-AVR-Microcontrollers-ATxmega64A1-ATxmega128A1_Datasheet.pdf.

[7] D. J. Bernstein. *Cache-timing attacks on AES*. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005.

[8] A. Bogdanov, D. Khovratovich, and C. Rechberger. "Biclique Cryptanalysis of the Full AES". In: *Advances in Cryptology – ASIACRYPT 2011*. 2011, pp. 344–371.

[9] M Burrows, M Abadi, and R. Needham. "A Logic of Authentication". In: *Proceedings of the Royal Society of London A*. Vol. 426. 1989, pp. 233–271.

[10] J. Deber et al. "How Much Faster is Fast Enough? User Perception of Latency Latency Improvements in Direct and Indirect Touch". In: *ACM CHI 2015*. 2015, pp. 1827–1836.

[11] *Free USB-IDs for shared use*. https://github.com/obdev/v-usb/blob/master/usbdrv/USB-IDs-for-free.txt. 2018.

[12] *Getting a Vendor ID | USB-IF*. https://www.usb.org/getting-vendor-id.

[13] GitHub user PRosenb. *Arduino EEPROMWearLevel*. https://github.com/PRosenb/EEPROMWearLevel.

[14] Hak5 LLC. *USB Rubber Ducky*. https://shop.hak5.org/collections/usb-rubber-ducky.

[15] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. 2008.

[16] A. Kerchkoffs. “La cryptographie militaire”. In: *Journal des sciences militaires* (1883).

[17] *KeyGrabber - Hardware Keylogger - Ordering*. https://www.keelog.com/cart/#prices. 2019.

[18] M. G. Kuhn. *Ideas for student projects*. https://www.cl.cam.ac.uk/~mgk25/project-ideas.

[19] M. G. Kuhn. *Part II Cryptography Lecture Slides*. https://www.cl.cam.ac.uk/teaching/1819/Crypto/crypto-slides.pdf. 2019.

[20] S. Kumar et al. “How to Break DES for €8,980”. In: *SHARCS’06 - Special-purpose Hardware for Attacking Cryptographic Systems*. Cologne, Germany, 2006.

[21] T. E. Kurt. *arduino-serial-lib – simple library for reading/writing serial ports*. https://github.com/todbot/arduino-serial/blob/master/arduino-serial-lib.c.

[22] B. Liviero. “Intel AES-NI Performance Enhancements: HyTrust DataControl Case Study”. https://software.intel.com/en-us/articles/intel-aes-ni-performance-enhancements-hytrust-datacontrol-case-study. 2014.

[23] O. Lo, W. J. Buchanan, and D. Carson. “Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)”. In: *Journal of Cyber Security Technology* 1.2 (2016), pp. 88–107.

[24] *MAX3421E USB Peripheral/Host Controller with SPI Interface*. https://www.maximintegrated.com/en/products/interface/controllers-expanders/MAX3421E.html.

[25] F. Mihailowitsch. *Detecting Hardware Keyloggers*. https://deepsec.net/docs/Slides/2010/DeepSec_2010_Detecting_Hardware_Keylogger.pdf. DeepSec 2010. 2010.

[26] NaCl Development Team. *NaCl Features*. https://nacl.cr.yp.to/features.html.

[27] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Standard. 2001.

[28] National Institute of Standards and Technology. *Guide to Bluetooth Security*. Tech. rep. 2012.

[29] R. Oppliger. *Contemporary Cryptography*. 2nd. 2011.

[30] V. Pavlik. *USB ACM - Kernel.org*. https://www.kernel.org/doc/Documentation/usb/acm.txt.

[31] C. Percival. “Stronger Key Derivation via Sequential Memory-Hard Functions”. https://www.tarsnap.com/scrypt/scrypt.pdf. BSDCan’09. 2009.

[32] B. Schneier. *Academic: Attack Trees - Schneier on Security*. https://www.schneier.com/academic/archives/1999/12/attack_trees.html. 1999.

[33] B. Schneier. “Applied Cryptography”. In: 1996, pp. 157–8.

[34] B. Schneier. *'Evil Maid' Attacks on Encrypted Hard Drives.* https://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html.

[35] *Trust 20623 Classicline Wired Full Size Keyboard for PC and Laptop.* https://amazon.co.uk/Trust-20623-Classicline-Keyboard-Laptop/dp/B017KDQ0LC.

[36] *USB Host Library for Arduino.* https://github.com/felis/USB_Host_Shield_2.0.

[37] R. Weatherley. *Arduino Cryptography Library: Generating random numbers.* https://rweather.github.io/arduinolibs/crypto_rng.html.

[38] *WhatsApp FAQ - End-to-end encryption.* https://faq.whatsapp.com/en/android/28030015/.

[39] Zoho Corporation. *6 reasons enterprises need to implement a USB security management system.* https://blogs.manageengine.com/desktop-mobile/desktopcentral/2017/04/12/6-reasons-enterprises-need-to-implement-a-usb-security-management-system.html. 2017.

[40] *avr-libc LICENSE.txt.* https://www.nongnu.org/avr-libc/LICENSE.txt.

[41] *distributed.net: Project RC5.* http://www.distributed.net/RC5/en. 2002.

[42] *uinput module - The Linux Kernel documentation.* https://www.kernel.org/doc/html/v4.16/input/uinput.html.

Appendix A

Protocol test vectors

This appendix contains an annotated dump of the messages that might be exchanged during setup and use of the system. These can be used as test vectors to implement a compatible product. All messages are shown in hexadecimal.

This section should be read in conjunction with the table of messages in subsection 3.2.1.

Multi-byte values are packed into messages in little-endian form. Each message begins with a single-byte *magic number* indicating what kind of message it is.

Setup

For the run detailed here, the IV value at boot is `0x0205`.

After booting, the device sends out `MSG_SETUP` over the serial line at regular (100ms) intervals:

```
aa 05 02 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The host responds with `MSG_SETUP_ACK`, which, aside from the magic number, must be all zero:

```
bb 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Operation

For this section, the shared keys are as follows:

k_1 (for encryption):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

k_2 (for authentication):

```
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Assume the user now presses the key 'A', with the left Control key also held down. Assume also that this is the seventh keystroke pressed since the device started up, so the counter has the value `0x06`.

The device packs the modifier key state and USB keycode into an `EVENT_KEYDOWN` message:

```
11 01 04 00 00 00 00 00 00 00 00 00 00 00 00 00
```

`0x01` is the value of the modifier field indicating that the left Control key is pressed. `0x04` is the USB HID key code for the key 'A'.

Encryption

To generate the keystream, the IV and counter are encrypted under k_1 with AES.

With IV `0x0205` and counter `0x06` (both 4 bytes), the keystream material is generated by encrypting the following input under k_1:

```
05 02 00 00 06 00 00 00 00 00 00 00 00 00 00 00
```

The resulting keystream material:

```
7f f8 e2 50 a7 14 d7 95 d6 ef 8e 1f 19 20 86 fe
```

The `EVENT_KEYDOWN` message is combined with the keystream material using XOR to give an encrypted message:

```
a1 a3 f7 ab b8 fc f6 23 66 60 f1 90 5f 7e f3 37
```

Authentication

The MAC is computed by encrypting the (encrypted) message under k_2) and discarding the last byte, to give a 15-byte value:

```
58 6f 5b 81 85 c8 bf 91 ad d9 2a 4d 6a d3 38
```

The message and MAC are combined into a `MSG_ENCRYPTED`:

```
dd a1 a3 f7 ab b8 fc f6 23 66 60 f1 90 5f 7e f3
37 58 6f 5b 81 85 c8 bf 91 ad d9 2a 4d 6a d3 38
```

This is sent to the host. Validation, decryption and unpacking of the message is a reversal of the steps detailed above

Rekeying

When the user presses the secure attention sequence (right-Ctrl+R), the device sends a `MSG_REKEY`, which, aside from the magic number, must be all zero:

```
cc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The host generates a random salt and password. For this run, let the password be:

```
ttx5ejj5im
```

and the salt:

```
e9 b9 34 c7 97 d7 0d dc e7 b1 52 e4 47 17 a9 13
```

The host sends the salt to the device in a `MSG_SALT`:

```
ee e9 b9 34 c7 97 d7 0d dc e7 b1 52 e4 47 17 a9
13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Both host and device compute new keys k_1 and k_2 from the password and salt, according to the PBKDF (Algorithm 1).

The resulting keys are:

```
k1:
18 ed c0 98 2c 81 69 49 ec a1 42 a1 87 4b 9d a8
fd ca d5 69 13 c9 89 63 a0 6c 88 86 70 3b 8b da

k2:
fd ca d5 69 13 c9 89 63 a0 6c 88 86 70 3b 8b da
e9 b9 34 c7 97 d7 0d dc e7 b1 52 e4 47 17 a9 13
```

Appendix B

Timing data

Run	Frames	Time (ms)
1	18	75
2	22	91.7
3	25	104.2
4	24	100
5	22	91.7
6	24	100
7	25	104.2
8	23	95.8
9	23	95.8
10	20	83.3
11	21	87.5
12	20	83.3
13	22	91.7
14	21	87.5
15	19	79.2
16	24	100
17	21	87.5
18	20	83.3
19	19	79.2
20	20	83.3

(a) Keyboard connected to PC via trusted-path system

Run	Frames	Time (ms)
1	17	70.8
2	12	50
3	14	58.3
4	13	54.2
5	11	45.8
6	10	41.7
7	12	50
8	12	50
9	14	58.3
10	11	45.8
11	11	45.8
12	12	50
13	12	50
14	10	41.7
15	9	37.5
16	14	58.3
17	10	41.7
18	10	41.7
19	8	33.3
20	11	45.8

(b) Keyboard connected directly to PC

Figure B.1: Raw timing data

Appendix C

Project proposal

Computer Science Tripos – Part II – Project Proposal

Trusted-Path Keyboard

Candidate 2346B, Churchill College

Originator: Dr M. G. Kuhn

10 October 2018

Project Supervisor: Dr M. G. Kuhn

Director of Studies: Dr J. K. Fawcett

Project Overseers: Dr A. Madhavapeddy & Prof. J. Daugman

Introduction

Input devices such as keyboards are a potential weakness in the trusted computing base of a modern PC. Hardware keyloggers allow an attacker to gain access to all information entered, while remaining essentially undetectable in software.

I will build a prototype device to provide an encrypted channel from a keyboard to a Linux PC, protecting against hardware keyloggers. I will also undertake a security analysis of the system and measure its performance.

Work to be done

The project consists of the following components:

1. Construction of a device which will receive plaintext input from an ordinary USB keyboard. It will transfer this input in encrypted form to a PC, where the input will be decrypted in software (detailed below). The Arduino platform (a development board for the ATmega series of microcontrollers) will be used as a base for this device.

   The device will be programmed in C, as is standard for the Arduino platform.

2. Development of software (the 'host software') to receive the encrypted input from the device in (1). On receiving and decrypting an input, the software will simulate an appropriate keystroke event using the `uinput` kernel module.

   The host software will also be written in C unless there proves to be a compelling reason to use another language. Reasons for this include ease of interfacing with the operating system (such as with `uinput`) and consistency with the embedded code.

3. Development of a protocol for the device and host software to communicate, This will involve researching and implementing a suitable symmetric-key cryptosystem, and later on implementing *password-authenticated key exchange* (PAKE) to establish a shared key.

   The side of of this protocol that runs on the device will need to work in an embedded environment with very limited memory.

   The PAKE implementation will involve the host software generating and displaying a password. The user enters this password into the device using the keyboard - these keystrokes are not sent to the host. Both sides will use this password to generate a shared key, which will be used for symmetric encryption. The device will need to indicate when it is in password-entry mode, for example by lighting an LED.

4. A qualitative security analysis of the device, host software and communications protocol. This will involve enumerating possible attacks and systematically assessing to what extent this system is vulnerable to them. This will entail detailed analysis of the protocol developed in (3).

5. Measurement of the time taken for a keystroke to pass through the system, looking specifically at the extent to which the cryptography increases latency.

   In order to measure the latency of the system as a whole, these measurements will be made in hardware, using a digital oscilloscope. This will involve running the host software on a Raspberry Pi single-board computer, which has GPIO pins that can be connected to the oscilloscope.

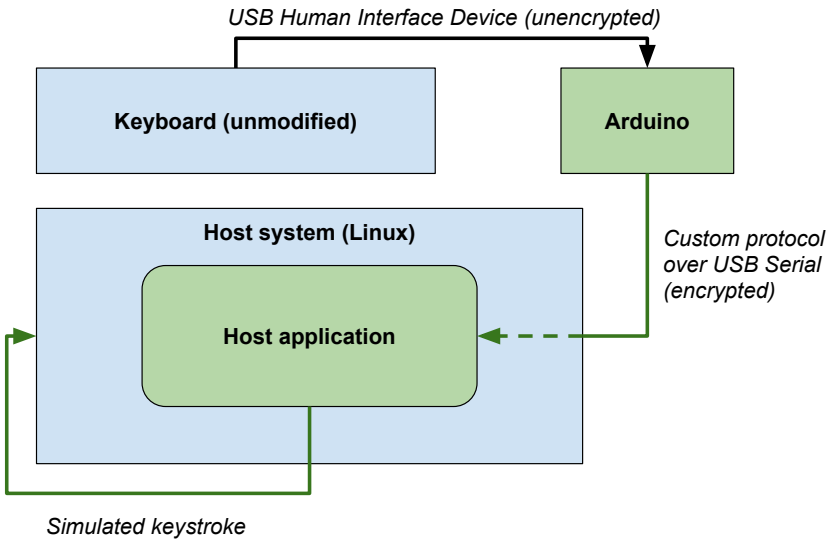

Figure 1: a diagram of the system. Parts in green are those I will develop (not including any extensions).

Starting point

- I have developed simple programs for the Arduino before, but I have never implemented cryptography or a non-trivial serial protocol.
- I have basic knowledge of C from Part IB and from a small amount of hobby programming.
- I intend to use the USB Communications Device Class protocol in Abstract Control Model mode for communication between the device and the host. This provides a virtual serial channel over which binary data can be sent, which avoids having to make any modifications to the USB stack on the host.

Resources required

- I will be using **my own personal computer** for development. My backup strategy is detailed below.
- An **Arduino Uno development board** and corresponding **USB host shield**. These are widely available and inexpensive. Alternatively, I may use an **Arduino Due** board, which has a more powerful microcontroller and can act as a USB host (to support a keyboard) natively.
- A **generic USB keyboard**. I have one of these already, and can purchase spares cheaply if necessary.

- Potentially some **basic electronic components** such as resistors, LEDs and breadboards (especially for some of the extensions). These are cheap and widely available.
- A **digital oscilloscope** to take timing measurements. Dr Kuhn is able to supply one of these.
- A **Raspberry Pi** single-board computer to act as the host in order to take timing measurements. I have several of these.

Success criteria

1. The system can be used to enter data into a PC using the keyboard.
2. The device and host software can establish a secure channel in the presence of an eavesdropper, for example by using password-authenticated key exchange.
3. A security analysis of the protocol is complete, discussing at least 3 potential vulnerabilities.
4. Basic timing data is collected, and this data is used to determine the effect (if any) of the cryptography on latency.

Possible extensions

- I may implement alternative encryption algorithms and/or key exchange protocols. In this case, I would extend the qualitative analysis and/or the timing measurements to cover these.

  For example, this might entail using a different symmetric cipher, or a key agreement protocol other than PAKE.

- As well as intercepting keystrokes, malicious hardware could inject its own, for example to execute commands to download malware. I may investigate this threat model and possibilities for mitigating it, such as authenticated encryption. Alongside this, I may investigate the feasibility of disabling other (presumably unauthenticated) connected input devices when my device is in use.
- Rather than acting as a USB host for an existing USB keyboard, I may modify the device to directly scan the switch matrix in the keyboard. This would bypass the onboard controller in the keyboard. Further timing analysis would then be performed to determine whether this significantly affects latency and/or jitter (variance in latency).
- I may extend the system to support the use of a mouse alongside or instead of a keyboard, providing the same security features.

Timetable

Planned start date is 19 October 2018.

1. **Package 1** (19 October - 1 November 2018)

   Order hardware (Arduino and USB shield) - these are available for delivery within 1 week.

   Begin to develop host software and investigate the use of `uinput`.

   Configure Arduino to receive keystrokes from keyboard and send (unencrypted) bytes to host software.

   Milestone: pressing a key on the keyboard triggers a simulated keypress on the host.

2. **Package 2** (2 - 15 November 2018)

   Begin to implement symmetric cryptography (with a fixed key) on device and host.

   Meet with supervisor to discuss suitable algorithms and operating modes.

   Milestone: device can send data to host software, symmetrically encrypted with a fixed key.

3. **Package 3** (16 - 29 November 2018)

   Research password-authenticated key exchange algorithms. Choose one and implement it. Add indicator for when device is in password-entry mode.

   Milestone: system behaves as in Milestone 2, but using a key agreed via a PAKE protocol.

4. **Package 4** (30 November - 13 December 2018)

   Slack time to recover in case previous packages are delayed.

   Write up details of implementation so far, for inclusion in dissertation.

   Milestone: implementation written up, possibly in bullet point form.

5. **Package 5** (14 - 27 December 2018)

   No work scheduled for this package - I intend to take a week off for Christmas and spend a week revising Paper 8/9 courses.

6. **Package 6** (28 December 2018 - 10 January 2019)

   Develop system to emit electrical signals for timing purposes. This includes recompiling the host software to run on a Raspberry Pi (a non-x86 architecture) and adding code to interface with the GPIO pins.

   Begin writing Introduction and Preparation sections of dissertation.

   Milestones: The device emits a signal immediately after receiving a keystroke from the keyboard; the host software runs on a Raspberry Pi and emits a signal on a

GPIO pin immediately after simulating a keystroke; a draft of the Introduction / Preparation chapters is complete as far as possible; these are sent to my supervisor for feedback.

7. **Package 7** (11 - 24 January 2019)

   Add functionality to device and host software to toggle whether encryption is used.

   Prepare progress report and presentation.

   Milestones: The cryptography can be toggled on and off with minimal effort on the device and host application. The progress report and presentation are complete or very nearly complete.

8. **Package 8** (25 January - 7 February 2019)

   Finish progress report (due 1 February) and presentation (date TBC).

   Learn how to use digital oscilloscope. Develop plan for how to use it to gather necessary timing data. Investigate suitable statistical tests for the data.

   Milestones: progress report submitted; presentation delivered or ready for delivery; plan developed for using oscilloscope to gather data.

9. **Package 9** (8 - 21 February 2019)

   Use oscilloscope to gather data. Perform statistical tests on data. Plot data graphically.

   Continue writing up: begin work on Evaluation section.

   Milestones: high-quality graphs of the timing data are produced suitable for inclusion in the project; a conclusion is reached as to whether the timing impact of the cryptography is statistically significant; draft of Evaluation section (reflecting work done so far) is complete.

10. **Package 10** (22 February - 7 March 2019)

    Begin security analysis of the system. Compile a list of possible attack vectors and investigate to what extent the system is vulnerable to them.

    Milestone: detailed analysis of at least 3 possible attack vectors added to Evaluation section.

11. **Package 11** (8 - 21 March 2019)

    Conclude security analysis.

    Continue writing up dissertation.

    Milestone: draft of the dissertation content is complete (some parts may be in bullet point form).

12. **Package 12** (22 March - 4 April 2019)

    Slack time.

    If slack time is not needed, work on extension(s) and/or continue writing up.

13. **Package 13** (5 - 18 April 2019)

    Continue writing up dissertation.

    Milestones: draft of dissertation in prose form is complete; graphs and diagrams are present in suitable quality for submission.

14. **Package 14** (19 April - 2 May 2019)

    Submit dissertation for feedback from supervisor and Director of Studies. Address any issues raised.

    Milestone: feedback addressed.

15. **Package 15** (3 - 17 May 2019)

    Finish writing up dissertation. Package source code into a ZIP archive for submission. Proofread and submit dissertation.

    Aim to submit by 10 May (1 week before deadline), as advised in the Pink Book.

    Milestone: dissertation and source code submitted.

Backup strategy

- Code (both for the microcontroller and the host application) will be stored under the Git version control system and synchronised daily with a private repository on GitHub.
- For the dissertation I will use the Overleaf online LaTeX environment. This combines a traditional TeX environment with Google Docs-style automatic versioning. To guard against the failure of this service, I will save a copy of the source to my filesystem, backed up by Dropbox, at least once every two days.
- In the event of a laptop failure, I will use my old laptop, which has the same version of Ubuntu Linux installed.