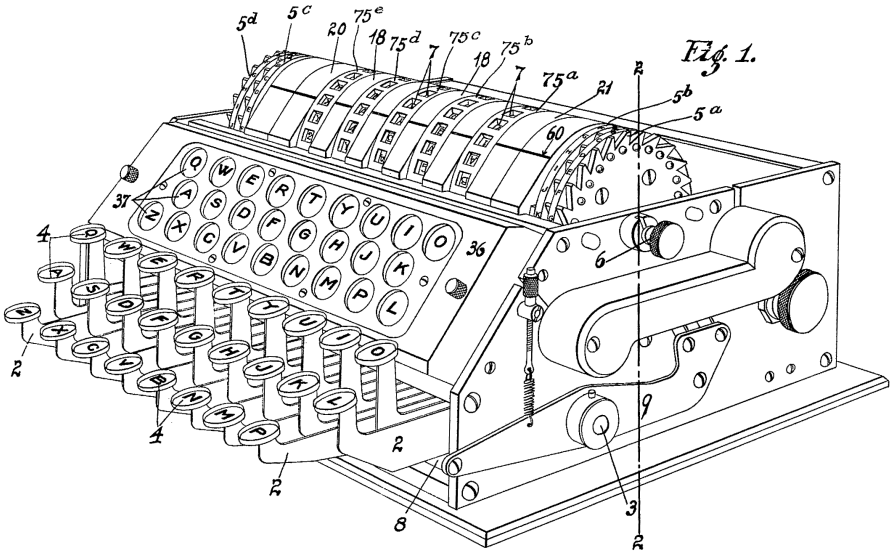
UNIVERSITY OF CAMBRIDGE

Harry Jones
hbgj2

TypeSafely
A Secure USB Keyboard



Harry Jones

Trinity College
University of Cambridge

Computer Science Tripos – Part II

June 2019

Declaration

I, Harry Jones of Trinity 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.

SIGNED /s/ Harry Jones

DATE 17 May 2019

I'd like to thank my supervisor and Director of Studies, Prof Frank Stajano, for all his support and guidance throughout the project.

This document is typeset in Minion Pro using LaTeX. The typographical look-and-feel is based on `classicthesis`, developed by André Miede and Ivo Pletikosić. The diagrams were produced using `msc.sty`, `tikz`, Prism 8, Sketch, and OmniGraffle.

The cover illustration comes from USPTO patent `US1683072A` "Electric code machine", which was filed in 1923.

TypeSafely – A Secure USB Keyboard

harryj.uk/projects/typesafely

Proforma

Candidate Number:	**2375C**
Project Title:	**A Secure USB Keyboard**
Examination:	**Computer Science Tripos, Part II, 2019**
Dissertation Word Count:	**11946**
Codebase line count:	**5350**[1]
Project Originator:	**Dr Markus Kuhn**
Supervisor:	**Prof Frank Stajano**

ORIGINAL AIM OF THE PROJECT

The protocol used by standard USB keyboards is insecure, allowing keylogging and the injection of malicious keypresses. This project aimed to design and implement a system, dubbed *TypeSafely*, to prevent such attacks by identifying and rejecting keystrokes from untrusted keyboards, as well as cryptographically securing and authenticating legitimate keystrokes.

The project would include designing a new cryptographic protocol, along with creating implementations of the protocol as a Linux device driver and as keyboard firmware. In addition, keyboard hardware and a GUI pairing interface would have to be developed to accompany the protocol implementations.

WORK COMPLETED

The project fully met the proposal's success criteria.

In the project, I designed a cryptographic protocol that gives the required security guarantees (making reference to Bluetooth's pairing protocol); created reference, microcontroller, and Linux device-driver implementations of the protocol; developed a GUI for pairing the keyboards; designed and assembled the necessary hardware to run the protocol with a physical keyboard; and performed post-implementation security analysis.

The end result is a usable, unintrusive, and secure system that could be realistically be deployed in the real world.

SPECIAL DIFFICULTIES

None.

1 Own code, excluding comments and blank lines

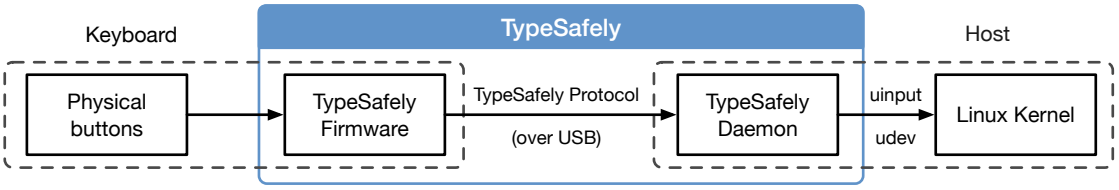
Contents

1 Introduction

The protocol used by standard USB keyboards is insecure. An attacker with physical access to a computer can insert a device which records keystrokes, or that spoofs keyboard input altogether. A skilled attacker can make these attacks invisible to both the keyboard and the host computer, meaning that a modification of the existing USB keyboard protocol is necessary in order to detect and prevent them. This project, TypeSafely, does just that.

TypeSafely is made up of three main components:

1. A protocol design and implementation that allows for encrypted and authenticated communication between USB keyboards and a host computer: the TypeSafely protocol
2. A Linux daemon and accompanying management interface to support TypeSafely keyboards on a host computer: the TypeSafely daemon
3. An implementation of the keyboard side of the protocol for a microcontroller: the TypeSafely firmware



In addition, I designed and assembled the necessary hardware to run the TypeSafely protocol on a physical keyboard.

1.1 MOTIVATION

USB peripherals effectively have privileged access to a system: they can send keyboard or mouse input, connect new network interfaces, or mount filesystems. However, the vast majority of USB devices communicate without any encryption or authentication. Despite this, most operating systems automatically connect new USB devices without requiring user approval. While the potential risk is somewhat mitigated due to the requirement of physical access, USB's privileged role and lack of security—combined with its ubiquity—has made USB-based attacks an attractive proposition for malicious actors.

For example, hardware USB keyloggers (which intercept keystrokes on-the-wire) have been commonly available for many years, but off-the-shelf keyboards still provide no protection against this attack. As hardware keyloggers can be invisible to the host computer, a user who wants to protect themselves must *physically* check for keyloggers.

However, the threat posed by USB implants is not limited to keyloggers. USB keyboard imitation attacks—where an attacker connects an inconspicuous device that sends keystrokes—can do anything a legitimate user can; the computer cannot distinguish these malicious

keystrokes from legitimate keyboard input. Barring physically searching for the implants, these attacks can't be prevented without authenticating all keystroke data.

Physically discovering such devices can be difficult; even the cheapest keyloggers are no bigger than a USB flash drive, making them difficult to notice when placed behind a desktop PC. Documents leaked from the US's National Security Agency in 2013 show that they can build a remotely-monitorable keylogger for just $30 [25], and even embed a complete man-in-the-middle (MITM) device into the a USB *connector* [24], making detection with the naked eye impossible. Researchers demonstrated proof-of-concept reimplementations of such devices in 2015, proving that they could be developed by civilians on a much smaller budget [29, 14]. In the past year, similar devices—such as the 'USBNinja' [33] or the 'O.MG cable' [21]–have become available for anyone to purchase, lowering the barrier for these advanced attacks.

Cryptographic encryption and authentication are necessary to protect users against the full range of attacks described above. TypeSafely's protocol specifies how to do so, and the accompanying implementations demonstrate its effectiveness.

1.2 RELATED WORK

1.2.1 CRYPTOGRAPHIC PROTOCOLS

There has been a great deal of research into designing cryptographic protocols that give the required security guarantees (confidentiality, integrity, authentication, authorisation). However, existing efforts are mostly aimed at protocols for networked or wireless devices. TLS, SSH, and Bluetooth are famous examples of cryptographic protocols that can give the required security guarantees. However, typical TLS configurations rely on a centralised trust model, requiring Certificate Authorities (CAs) to certify a server to be running on a particular domain.

SSH is decentralised, but has no convenient pairing mechanism for keyboards: the client must choose to trust the server's public key fingerprint on the initial connection.

Bluetooth's model is a better fit for the project: it's decentralised, and includes a simple peer-to-peer pairing process. Bluetooth supports many different pairing and I/O methods, some of which are insecure or inefficient. However, some past design decisions could be optimised in a newly-designed protocol. A more in-depth analysis of the protocols is given in section 2.3.

1.2.2 PROTECTION AGAINST USB ATTACKS

The USB Implementer's Forum (USB-IF) has a specification for authenticating USB Type-C devices [32]. This mainly aims to protect against counterfeit devices, so it doesn't encrypt or authenticate any data that the device sends after enumeration. Therefore, it doesn't provide protection against keylogging or modifying the keypresses reported by a legitimate keyboard.

GoodUSB [28] is a research project that protects against attacks where a device appears to have one role (e.g., a USB flash drive) and yet performs another (e.g., a keyboard). It does this by having the user input their expectation of what the device is, which is used to limit

the device to its expected role. This would prevent some attacks, however doesn't prevent keylogging or modifying keypresses from a legitimate keyboard.

Researchers at IBM designed UScramBle [20], which uses encryption to protect against passive eavesdropping attacks on USB devices. This would circumvent currently available keyloggers, but doesn't prevent MITM attacks from intercepting keypresses.

Cinch [1] is a system that isolates USB devices from the OS's USB stack by keeping the host OS in a VM. USB devices are connected to a separate 'sacrificial' VM that forwards USB packets to the host OS, filtering out attacks against known vulnerabilities in the OS. This approach can also allow for authentication and encryption of USB traffic in a way that's transparent to the host OS. However, this doesn't provide a solution for pairing the device—the devices have to know to trust each other in advance, for example via the use of a trusted CA. A pairing process could feasibly be built on top of Cinch, but without one it would be difficult to deploy Cinch in the real world.

	Passive Keylogging	Active Keylogging	Keyboard Spoofing	Keypress Modification	General?
USB-IF's Auth.	N	N	Y	N	Y
GoodUSB	N	N	Y	N	Y
UScramBle	Y	N	N	N	Y
Cinch	Y	Y	Opt.	Y	Y
TypeSafely	Y	Y	Y	Y	N

Table 1.1: Comparison of the types of attack protected against by different solutions. 'Active Keylogging' refers to a MITM attack that repeats/forwards data packets without modification. 'Keyboard Spoofing' is where an attacker makes use of USB device masquerading as a keyboard. 'Keypress modification' is the alteration of keypresses from a legitimate keyboard. The 'General?' column indicates whether or not the solution works for other types of devices, rather than just keyboards.

2 Preparation

This chapter details the steps I took to determine how best to develop my implementation. It first describes the USB keyboard protocol, possible attacks against it, and my research into how to debug USB communications. Next, I describe the protocol requirements, which are derived from a model of the threats faced. Finally, I describe my research into existing cryptographic protocols and potential hardware, before outlining my development strategy.

2.1 Analysing USB

To develop effective security enhancements for USB keyboards, it was important to understand how they communicate with host computers. I'd had limited prior experience developing with, so this information was an important starting point. These details were gathered from the core USB 2.0 specification [31], a surprisingly clear and terse 650-page document.

USB is a packet-based protocol that operates between a host and a peripheral device. When a device is connected, the host *enumerates* it, fetching various *descriptors* from the device. These give information such as its serial number and capabilities. A device can report multiple capabilities by listing multiple *interfaces*; for example, a device might report both a keyboard interface and a mouse interface. The host then loads the appropriate drivers for the device. These communicate with the device by sending packets back and forth to individual unidirectional *endpoints* on an interface. Any data can be encapsulated in these packets, making USB a very flexible protocol.

2.1.1 USB keyboard protocol

To improve interoperability, USB specifies some standard *device classes*, each with standardised configuration, behaviour, and packet formats. The "Boot keyboard" specification is implemented by virtually all keyboards, and is documented in *Device Class Definition for Human Interface Devices* [12]. The specification is designed to give keyboards a common, minimal protocol so that even space-constrained pre-boot environments (e.g., BIOS) can use USB keyboards. The protocol specifies that hosts will query keyboards for the current set of pressed keys at regular intervals, as shown in Figure 2.1.

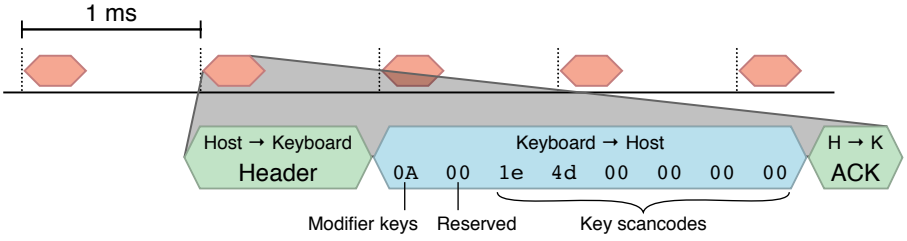


Figure 2.1: Diagram showing the structure of each USB keyboard transaction

As these packets are sent without encryption or authentication, an attacker can eavesdrop on or modify the packets on-the-wire relatively easily.

2.1.2 USB KEYLOGGERS

Hardware USB keyloggers record keystrokes by eavesdropping a keyboard's data lines. They can be invisible to the host computer, and typically come in a discreet form factor—about the size of a USB flash drive—with a USB port at one end and a USB plug at the other. One such example is shown in Figure 2.2.

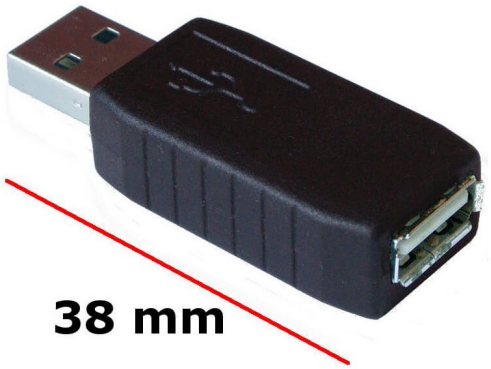


Figure 2.2: A USB Keylogger. This specific device provides 16 *MB* of onboard storage. Image and annotation source: www.spy-shop.org

Using a basic encryption protocol, without authentication, would thwart eavesdroppers, but more advanced keyloggers could overcome this; a keylogger that can change packets on-the-fly could insert itself into the connection, decrypting and re-encrypting packets to observe the keypresses. Therefore, in order to give anti-keylogger guarantees, cryptographic authentication is necessary.

2.1.3 USB KEY INJECTION DEVICES

These maliciously send keystrokes by presenting a keyboard interface to the computer. Since devices self-report their capabilities, even devices that visually appear to have a completely different purpose can send keyboard input. This makes *any* untrusted USB device a potential threat. The most well-known device is the *USB Rubber Ducky*, a cheap device that comes disguised as a flash drive.

The scope of the impact of these attacks is wide: since these devices appear to be a standard keyboard, the host OS trusts these devices to make any keyboard input an actual user could. One common attack is to run a malicious script in a terminal to give persistent access.

I'm not aware of any devices that modify keypresses from legitimate keyboard—there's no need, since the above attacks are less complex. Nevertheless, it is a potential attack vector that should be considered.

2.1.4 DEBUGGING USB

I anticipated that at some point it'd be necessary to gain insight into the low-level USB communications going on over-the-wire. I discovered in my research that there are many different solutions:

- Software packages, such as `usbmon`, log communications visible at specific software layers of the USB stack. However, software solutions can't give the full picture: the USB hardware invisibly performs some communication and error-handling itself
- Specialised hardware. Dedicated protocol analysers exist, but are not cheap (£100s). FPGA-based solutions exist, but a suitable FPGA was not easily available to the author.
- Logic analysers. These can work well when combined with software to decode USB signals. Figure 2.3 shows how a logic analyser might be set up to record signals from a USB cable. Sadly, cheap logic analysers can only record USB traffic for a few seconds due to the high speed of USB and their small buffers.

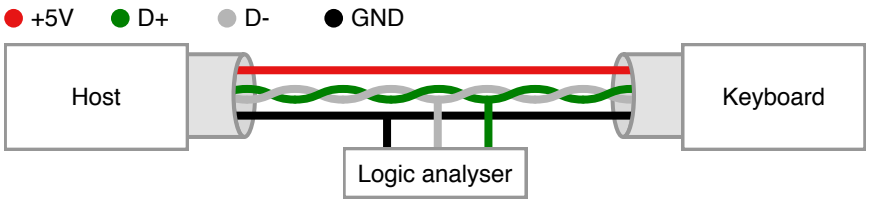


Figure 2.3: Diagram showing how a logic analyser can be connected to record data transmitted over a USB cable.

I decided to use a logic analyser for this project due to their price and availability, and used the `sigrok` package to decode USB packets.

2.2 ATTACKER MODEL

It is important to decide exactly what an attacker's capabilities are to determine requirements for a secure system. It's impossible to create a panacea for all attacks[1], but it *is* possible to create a system that's secure within some predefined boundaries.

The environment I targeted for the project was a security-conscious organisation that wants to protect itself against an attacker that might be able to get temporary physical access to their offices—by bribing a cleaner, for example. Therefore, in my model, an attacker is able to connect their own arbitrary USB devices to the user's computer, or to intercept and modify any communications happening with existing USB devices on-the-fly (e.g., by hiding a dongle behind a desk). They may also temporarily connect a TypeSafely keyboard to their own host, to try and break the system with their own software.

I assume that the user has the TypeSafely daemon running, and that their computer is initially free from both bugs that may be triggered over USB, and malware—otherwise an attacker

1 Simple proof: XKCD 538

could make use of keylogging malware to defeat the security of the system. I also assume that the attacker doesn't know the user's login password.

In the project, I don't try to prevent Denial of Service (DoS) attacks: an attacker with physical access could just cut the wires anyway, so preventing these is not feasible. I also don't try to protect against *software* keyloggers: their mechanism is entirely different that of hardware keyloggers, and any attacker who can install a software keylogger on a computer probably has a high degree of control over the computer anyway.

I don't try to defend against an attacker tampering with the keyboard's internals, for example by extracting a private key from onboard storage. While it's possible to protect against such attacks with tamper-evident hardware, attacks of this nature are unlikely under my threat model, and are difficult to prevent within the limitations of a Part II project. By the same reasoning, RF-based and power measurement/glitching attacks are also not considered.

2.2.1 REQUIREMENTS ANALYSIS

The requirements for the protocol are largely derived from the attacker model: the protocol has to be able to prevent keylogging and key injection attacks in the face of an attacker who has physical access. Therefore, the protocol has to guarantee confidentiality, integrity, authentication, and authorisation for USB keystrokes.

As it's designed for a 'normal' user, its use should not be a burden and should not require specialist skills or knowledge. My implementations should run on a microcontroller embeddable inside a keyboard, and shouldn't have an adverse impact on the host PC's performance.

Additionally, the protocol should be implementable in a variety of languages and make use of standard cryptographic primitives, to allow for ease of implementation for different environments (e.g, a Python implementation, one for a microcontroller, etc.).

2.3 CRYPTOGRAPHIC PROTOCOL RESEARCH

History has shown that designing secure cryptographic protocols is hard. With such a history of flaws, it'd almost be arrogant to try to create a secure cryptographic protocol from scratch. Instead, I decided to base my protocol on a well-studied and well-tested protocol.

I considered TLS, SSH, and Bluetooth's pairing protocol, all of which give the required security guarantees of confidentiality, integrity, authentication, and authorisation. As described in subsection 1.2.1, I found that Bluetooth's decentralised pairing system was the best fit for the project, and so I looked into it more in depth.

2.3.1 BLUETOOTH PAIRING PROTOCOL

Bluetooth's pairing protocol is built on a decentralised model, and allows devices with various input/output capabilities to authenticate and communicate with each other securely. Having been designed for use by nontechnical users, it also specifies simple pairing methods that

minimise risk of security failure due to user mistakes. The classic example is the 6-digit PIN system, where the user enters the PIN displayed on one device into the other. Other examples include the numeric comparison method, where the user confirms that the two devices display the same number onscreen. In this project, I chose to focus on the PIN entry method, since it fits the input capabilities of a keyboard the best. This is documented in-depth in section 3.2.

Bluetooth supports a variety of device types and use-cases, and is highly backwards-compatible. Because of this, many things could be optimised in a new, specialised implementation. Therefore, I chose to use parts of it as the basis for my own cryptographic protocol, rather than implementing it outright.

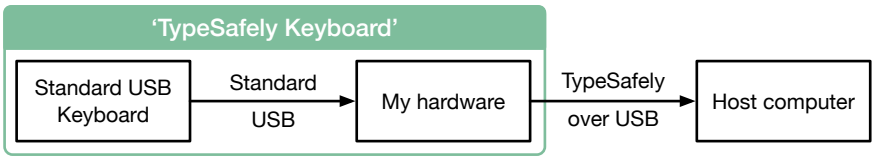
The *Bluetooth Core Specification* [11] is a 2,822-page document that specifies the parts of Bluetooth that most devices implement. In advance of the implementation, I studied the security-related sections and established the purpose of each part of the pairing protocol. This allowed me to remove or tweak elements in my own implementation, while still making use of the battle-tested elements of the protocol.

I also looked at past vulnerabilities in Bluetooth and its implementations. These served as warnings about areas that were overlooked or mis-implemented in the past, and that I should keep in mind for my own design and implementation.

2.4 HARDWARE RESEARCH

To create a complete implementation, the project would need some physical hardware to act as a keyboard. To demonstrate the project working end-to-end while minimising implementation time, I decided to create a device that could be inserted between an existing USB keyboard and computer.

Nevertheless, conceptually the hardware should all be thought of as being inside a standard keyboard, as this is how it would be deployed to a real user.



There's a wide range of microcontroller platforms available to run the code, including the venerable Arduino series. However, the microcontrollers used by much of the Arduino series are relatively underpowered by modern standards, and the programming environment doesn't scale to larger projects well.

In the end, I chose an ARM-based STM32F415 board, the 1Bitsy [13] because of its high clock speed and built-in hardware acceleration of many cryptographic operations, which would give flexibility in creating a fast protocol implementation.

The MITM-like approach means the system has to be able to act as both a USB host and a USB device. The STM32F415 supports acting as both simultaneously, but I elected to only use it as a USB device and have an *external* USB host module for simplicity. Acting as a USB host

can be complicated, since the downstream USB topology to the keyboard can be complex. I decided to use a HobbyTronics USB Host board [16] for this purpose.

I chose not to include a Trusted Platform Module (TPM). TPMs perform cryptographic operations without letting secret keys ever escape the TPM. They often include tamper-protection, making extracting secret keys difficult. However, an attacker with access to the keyboard's internals could still use the TPM to authenticate and encrypt anyway. Preventing this—by tamper-proofing the entire keyboard—makes TPMs largely redundant.

2.4.1 PROGRAMMING ENVIRONMENT

I wanted to write my microcontroller firmware in a way which would allow for using a different microcontroller in the future. However, writing portable code for ARM microcontrollers is difficult: only a fraction of the C standard library is available, and there's no standardised way to interact with hardware such as GPIOs. At a low-level, this interaction normally happens by reading from/writing to special memory locations. The location and format of these varies between different models, let alone different vendors. To help remedy the situation, there's a number of different Hardware Abstraction Layer (HAL) libraries that give an API to access these features.

In this project, I chose to use libopencm3 [18], an open-source HAL supporting microcontrollers from many vendors. Unfortunately, libopencm3's API for accessing each microcontroller feature can vary from microcontroller to microcontroller. I chose to create my own library—using libopencm3 under-the-hood—to give a single high-level interface for interacting with microcontroller features, separating the platform-specific code from the main codebase.

2.4.2 USB INTERFACE

As discussed earlier, USB devices expose a hierarchy of configurations and interfaces, with individual endpoints for transmitting/receiving data at the lowest level. From analysis of the USB specification [31] and existing USB devices, I decided to implement my secure keyboard protocol by creating a custom USB interface, in addition to the standard USB Boot keyboard interface, under a single USB configuration. This topology is shown in Figure 2.4.

Alternatively, I could have put the Boot interface and the Secure interface each under their own configuration. However, since only one configuration is active at a time, and devices can't change their own configurations, if the TypeSafely daemon exits (e.g., if a user reboots into BIOS) the keyboard couldn't revert itself to the insecure mode. Adding the required endpoints under the Boot interface might have worked, but would have been a violation of the USB specification.

As Figure 2.4 shows, the secure keyboard needs both an input and an output endpoint to permit bidirectional communication. I used Bulk endpoints because interrupt endpoints only allow data transfer at fixed intervals. This means pairing messages can be exchanged more quickly, but that USB no longer guarantees keypress packets will be delivered at exact intervals. In practise, I didn't find this to be an issue.

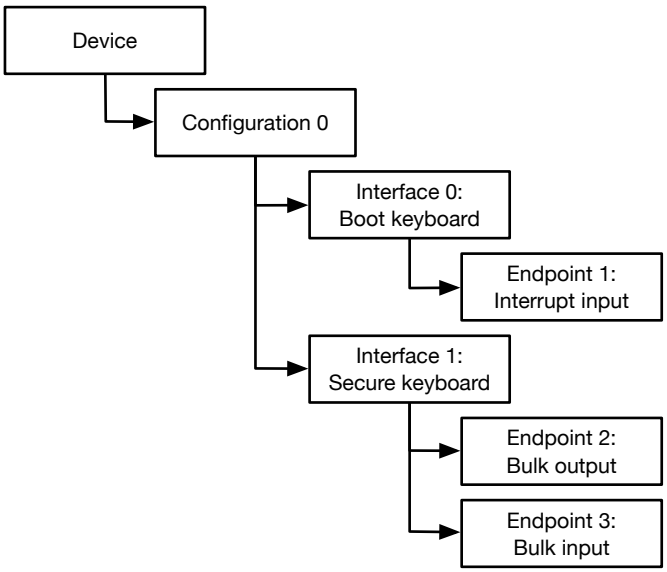


Figure 2.4: Diagram of USB topology, showing where the custom secure interface fits in

2.5 DEVELOPMENT STRATEGY

I chose to implement this project in phases:

1. Choose and assemble the necessary hardware to physically run the microcontroller implementation
2. Design a specification for the protocol, clearly detailing which aspects have been adapted from Bluetooth and how.
3. Develop a clear reference implementation of the protocol in Python. I chose Python because it's syntax and sizeable standard library allow for terse, readable code. A reference implementation would allow me to find a good code structure for my 'real' implementations, and provide a reference to validate them against.
4. Create a C++ implementation of the protocol, with an accompanying test program. I chose to use C++ due to its portability and some of its additional features over C, such as type-checked array sizes, that lend themselves to creating clear, safe code. However, I would limit myself to a small subset of C++ features to keep the code simple and efficient, even on a microcontroller.
5. Develop the actual microcontroller and host implementations.

I had very little previous experience with USB, C++, and larger-scale embedded software development, so I made sure to budget extra time to deal with any development issues that I might encounter.

2.5.1 STARTING POINT

As per the project proposal, the codebase will be developed from scratch. I chose and assembled my own hardware without using any existing reference. As described earlier, the cryptographic protocol was based on Bluetooth's pairing protocol, but this is detailed in-depth in my implementation (see section 3.2).

Implementation 3

In this section I describe the work I did to create the system. I first describe my hardware assembly and protocol design. Next, I describe my reference and final implementations of the protocol, the platform-abstraction library I use to make my microcontroller implementation terse and portable. Finally, I detail the actual host and microcontroller codebases.

3.1 KEYBOARD HARDWARE

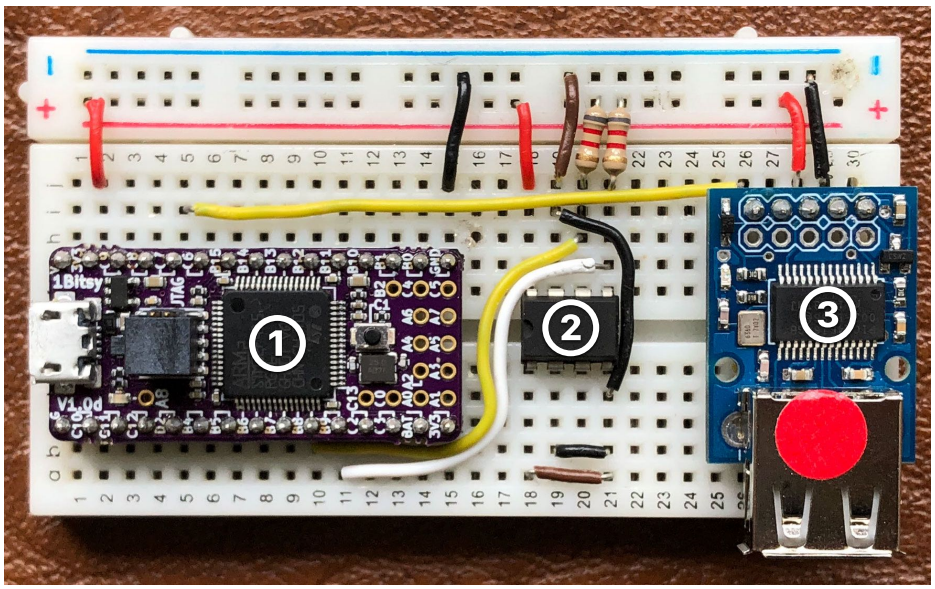


Figure 3.1: Photo of completed hardware assembly. (1) is the main 1Bitsy microcontroller board, (2) is an EEPROM chip, and (3) is the USB keyboard host module.

In order to use the project as a real keyboard, it was necessary to build some physical hardware to run the code. As discussed in section 2.4, for the project I chose to create a device that uses an existing keyboard for input. Conceptually, the overall 'TypeSafely keyboard' is made up of both my custom hardware and the input keyboard.

My design is centred around a STM32F415-based board, the 1Bitsy [13]. This acts as the USB device, but I used a HobbyTronics USB Host board [16] as a host. This sends the raw USB scancodes over a serial link to the 1Bitsy.

I used a SEGGER J-Link Mini Education to program and debug the 1Bitsy via its onboard SWD connector. Using the accompanying software, I was then able to program and debug the microcontroller over remote-GDB using my preferred IDE.

I also chose to use an external EEPROM chip. This allows data to be persistently stored much more quickly and easily than using the STM32F415's onboard flash, which requires slower, multi-kilobyte block-level writes and has a limited write-cycle lifespan.

Figure 3.1 shows a complete hardware assembly. The 'input' keyboard would connected to (3), and (1) plugs into the host computer via micro-USB cable.

The 1Bitsy has a LED, which TypeSafely requires to indicate the current connection state: rapid flashing indicates an insecure connection, pulsating indicates PIN entry mode, and a steady light indicates a secure connection. In a 'real' TypeSafely keyboard, a similar LED must be included, or a user might be tricked into using the keyboard in an insecure mode. The different modes are shown in Figure 3.2.

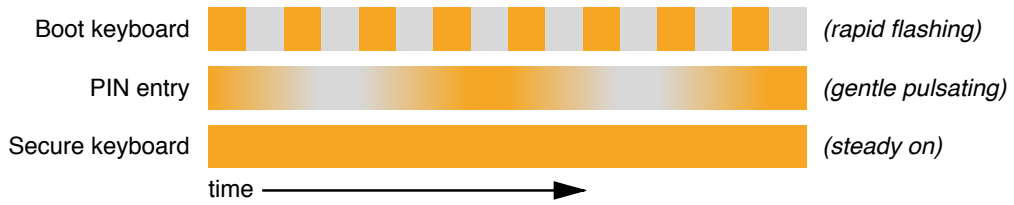


Figure 3.2: Diagram showing how the on-board LED's brightness is used to represent different keyboard modes.

3.2 PROTOCOL

The TypeSafely cryptographic protocol is the core of the project. It defines the way that the keyboard and host securely authenticate and communicate with each other. The protocol has to fit the requirements given in subsection 2.2.1.

3.2.1 PROTOCOL SPECIFICATION

The protocol is made up of many stages, with predefined progressions between them. In the event of an error, such as mismatched cryptographic check values, the connection will jump back to an earlier stage and another attempt will be made.

The pairing stages of the protocol are derived from Bluetooth's pairing specification (see subsection 2.3.1). For clarity, the protocol diagrams highlight the items that only appear in **Blue**tooth in **blue**, only in my protocol in **red**, and shared items in **black**. Some variables have also been renamed for clarity.

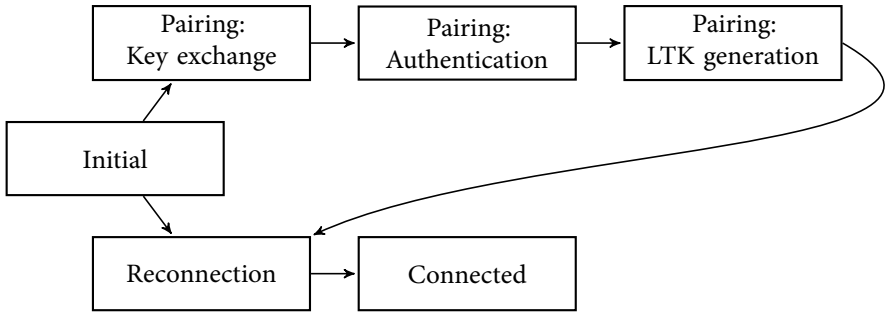


Figure 3.3: Diagram showing connection flow

3.2.1.1 *Initial connection stage*

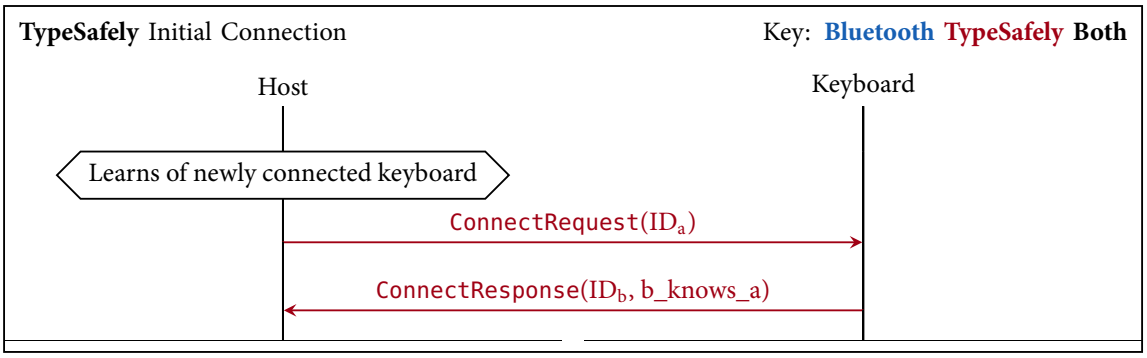


Figure 3.4: Sequence diagram for protocol stage 1 — Initial connection stage

The devices first recognise each other by their IDs. A device's ID only changes when the user resets it—which they might do before selling it, for example.

When the host's USB stack first detects a TypeSafely-capable keyboard, it sends a `ConnectRequest` message. This message informs the keyboard that the host is also TypeSafely-capable, and passes the host's ID, ID_a, to the keyboard.

The keyboard then checks if it's paired with ID_a, and sets the value of boolean *b-knows-a* if so. It responds with a `ConnectResponse`, containing this boolean and its own ID (so the host can know if it has a valid pairing with the keyboard).

Next, if either device didn't know the other, the host initiates a new pairing, documented in subsubsection 3.2.1.2. Otherwise, the host jumps to the Reconnection stage, documented in subsubsection 3.2.1.5.

3.2.1.2 *Pairing: Key exchange stage*

Key exchange is the first part of the pairing process. This stage is a textbook Elliptic Curve Diffie-Hellman (ECDH) exchange, using the Curve25519 curve [7]. This process allows the two sides to derive a shared key, DHKey, that an eavesdropper cannot obtain. Note that an attacker that can modify messages could still insert their public keys into this exchange, and so authentication of the devices at either end is still necessary later.

I chose to use Curve25519, rather than the more established P-256 curve that Bluetooth uses, because Curve25519's implementation can be both smaller and faster, whilst also having fewer implementation pitfalls. There's also some suspicion among cryptographers that P-256's parameters may have been chosen to 'backdoor' the curve—as has happened before [8].

First, each device generates its own ECDH public key PK_x and private key SK_x. These keys are discarded after the pairing process. The host then transmits an `InitiatePairingRequest`, containing its public key and *a-knows-b*. Bluetooth doesn't include *a-knows-b*, but TypeSafely introduces it to allow both sides to know if just one party remembered the other. This isn't a security feature, but could be used as an indicator to the user of why a pairing is needed.

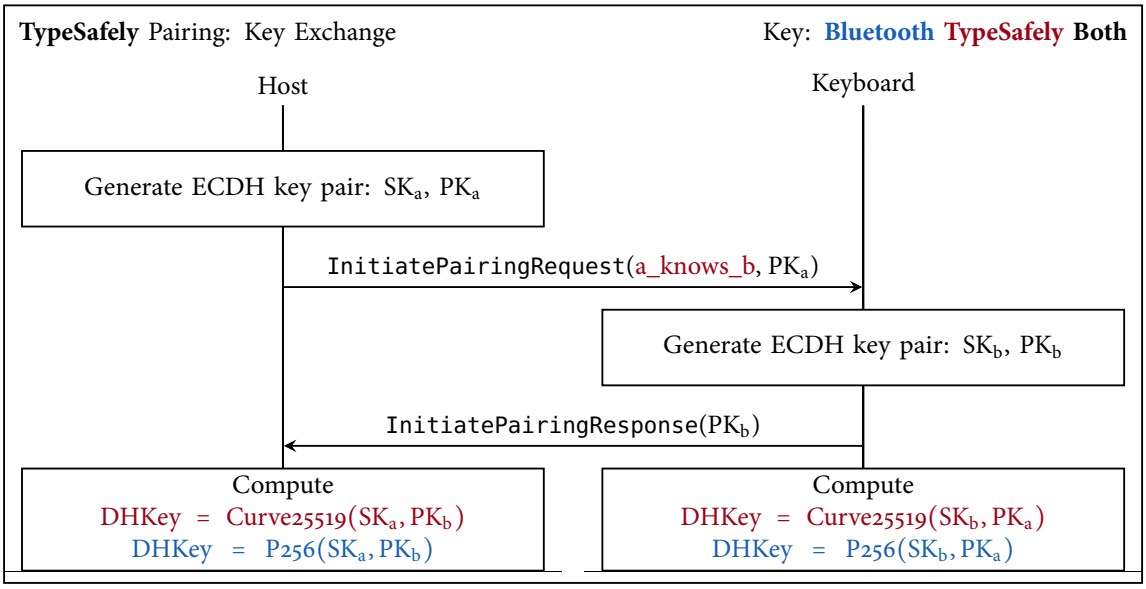


Figure 3.5: Sequence diagram for protocol stage 2 — Pairing: Key exchange stage

The keyboard then responds with an `InitiatePairingResponse` containing its public key. Finally, the two both derive DHKey from their respective public and private keys.

3.2.1.3 *Pairing: Authentication stage*

The devices now must prove to each other that they are the devices the user is interacting with, and not an attacker. This process is based on Bluetooth's *Passkey Entry* authentication method.

The host first randomly generates and displays a n-bit PIN onscreen, then sends an empty `BeginAuthenticationRequest` to the keyboard. Once the user enters the PIN, the keyboard sends an empty `BeginAuthenticationResponse`. The keyboard doesn't transmit PIN keypresses, preventing a keylogger from observing the PIN. Most of the time, a 6-digit PIN is used, giving $n = 20$: $2^{20} > 999999 > 2^{19}$.

At this point, both sides know the PIN r, but each must prove to the other that they know r without revealing it. To do this, they each generate a *commitment* value C_x. These values are calculated using f4, whose definition comes from the Bluetooth specification:

$$\mathrm{f4}(\mathrm{PK}_{\mathrm{self}}, \mathrm{PK}_{\mathrm{other}}, N_x, r_x) = \mathrm{AES\text{-}CMAC}_{N_x}(\mathrm{PK}_{\mathrm{self}} \,||\, \mathrm{PK}_{\mathrm{other}} \,||\, r_x)$$

That is, the commitment values are the AES-CMAC of the concatenation of the two public keys and the PIN number r, keyed by a random number N_x. The MAC uses N_x as a key because it has 128-bits of entropy; r is only 6 digits long, so using it as the key would allow an attacker to recover r by trying values until they find the one that gives the same commitment as the other side sends. Brute-forcing the 128-bit N_x isn't computationally feasible, and neither is finding a N_x that would make an incorrect C_i value correct after discovering r.

Exchanging commitments at this stage forces each device to commit to a value of r, meaning that one side can't pretend to know r by repeating the r revealed to it by the other later on. The commitment values are secure for this purpose because:

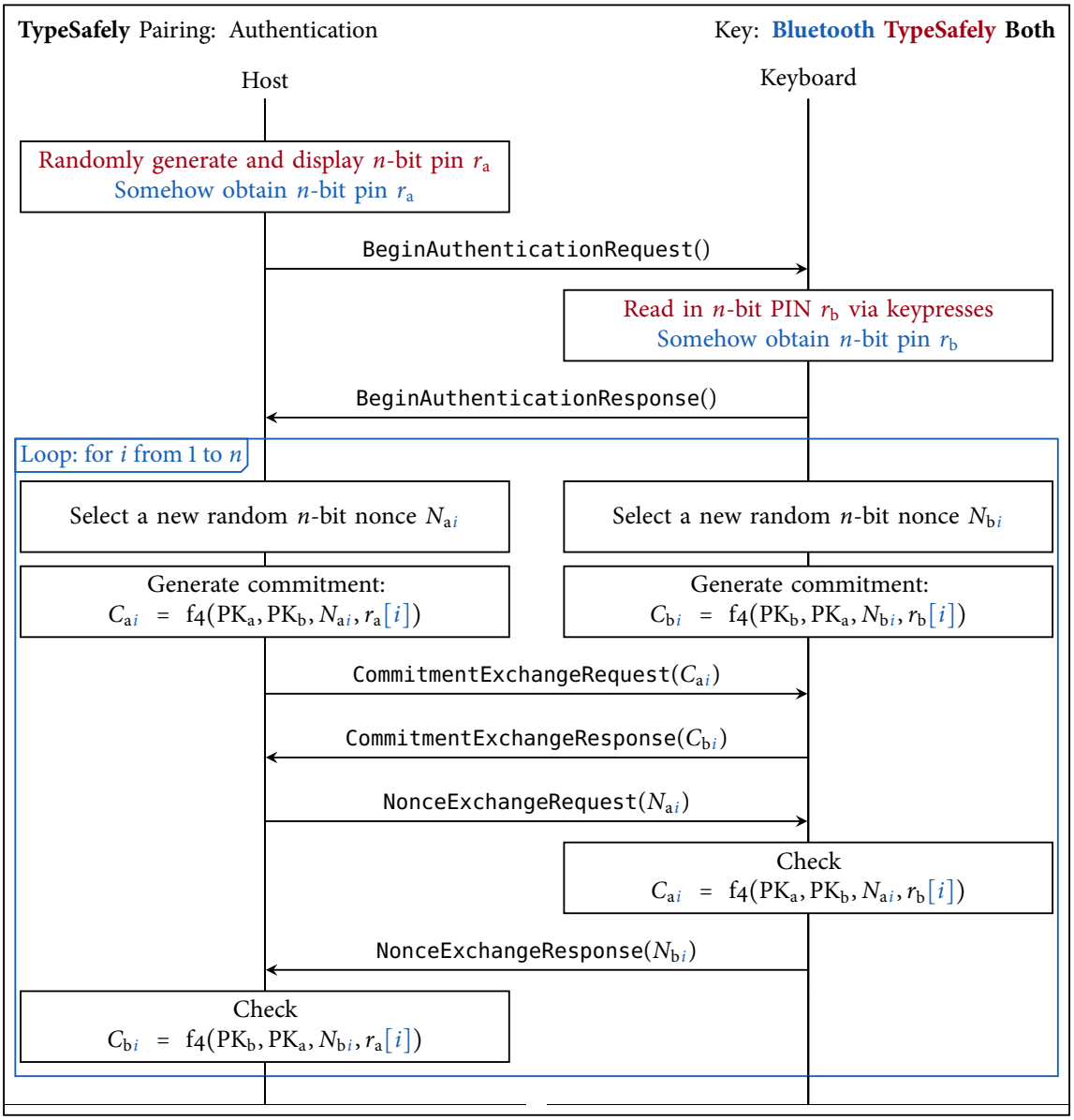


Figure 3.6: Sequence diagram for protocol stage 3 — Pairing: Authentication stage. $x[i]$ is the i-th bit of x.

- The N_x values aren't revealed until both commitments have been sent, meaning r can only be feasibly brute forced after both sides have committed to a value of it.
- The two sides' commitments are different—the order of the keys input into f4 is swapped—meaning that one side's commitment can't just be sent back to the other.

If either commitment check fails, the device sends an error and restarts pairing from the beginning (including generating new keys). Otherwise, both devices can be confident that $r_a = r_b$, and therefore that they are talking to the legitimate other device. Note that N_a is checked before N_b is sent, meaning that the keyboard will never reveal the PIN if the other side doesn't already know it. This prevents a MITM from obtaining the PIN the user types then forwarding it on to the host.

While a MITM attacker could transparently pass these commitments along, if they had swapped their public key $\mathrm{PK}'_{\mathrm{a}}$ into the connection earlier, the two legitimate devices would calculate different commitments:

$$C_a = \mathrm{f4}(\mathrm{PK}_{\mathrm{a}}, \mathrm{PK}_{\mathrm{b}}, N_{\mathrm{a}}, r_{\mathrm{a}}) \neq \mathrm{f4}(\mathrm{PK}'_{\mathrm{a}}, \mathrm{PK}_{\mathrm{b}}, N_{\mathrm{a}}, r_{\mathrm{a}}) = C'_a$$

Therefore, this process also ensures that the public keys haven't been tampered with, meaning `DHKey` was generated between 'legitimate' devices.

Bluetooth performs the commitment/nonce exchange process n times, generating the confirmations using each bit $r_x[i]$ of the full n-bit PIN r_x in turn. This has the benefit of only leaking 1 bit of the PIN to the other party each round, rather than revealing the entire PIN at once. This is desirable in Bluetooth, because Bluetooth doesn't specify how the PIN should be generated, meaning it might be re-used in different pairing attempts. Therefore, without this an attacker could force a legitimate pairing to fail after the nonces have been revealed but still learn the full PIN. If the pairing is re-tried with the same PIN, the attacker could successfully prove they know the PIN. This attack, and possible countermeasures, was discussed by Barnickel et al. [5]. The attack isn't a concern in TypeSafely because it specifies that the PIN must be chosen completely at random, precluding PIN reuse. Therefore, there's no need to have the $(n-1)$-extra round trips required by only committing to a single bit at a time, so the entire PIN is committed to at once instead.

3.2.1.4 *Pairing: LTK generation stage*

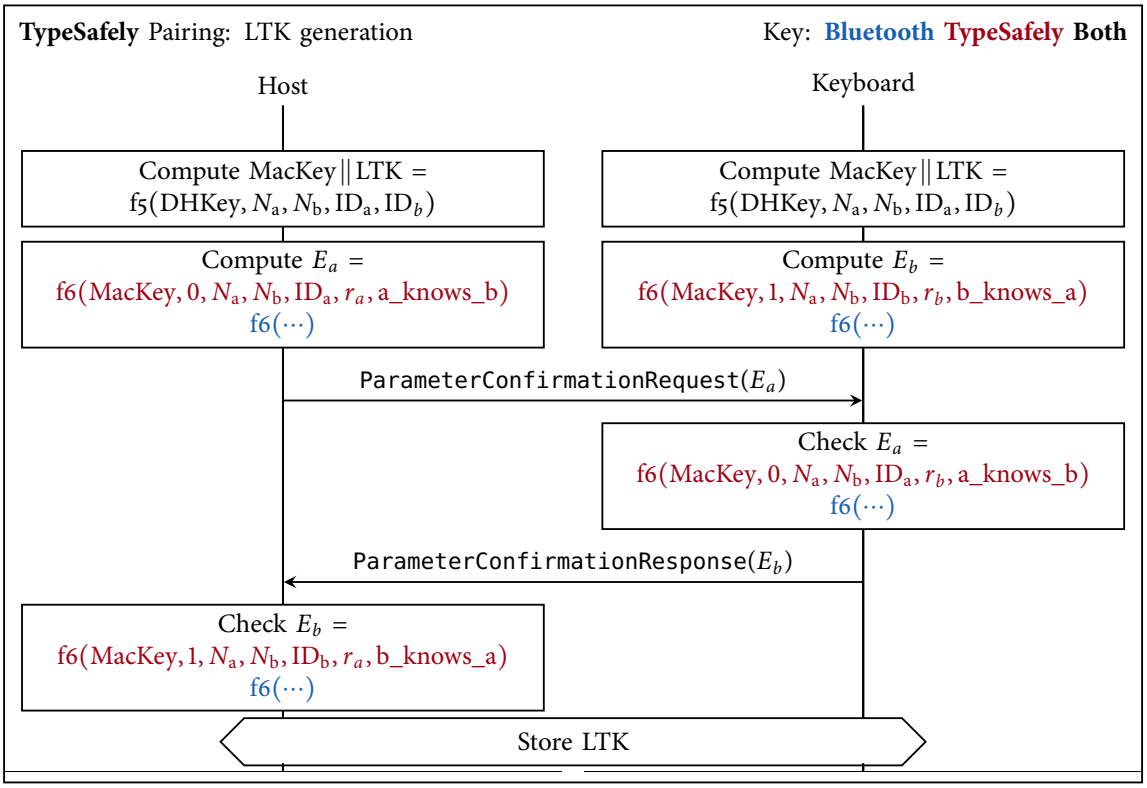


Figure 3.7: Sequence diagram for protocol stage 4 — Pairing: LTK generation stage

The devices now authenticate the previously-sent but non-authenticated parameters: their IDs, and the *x-knows-y* values. Without checking these parameters, an attacker could force a new pairing to be made without either side being aware of the interference, which isn't insecure, but isn't desirable.

First, each side uses f5 to generate a `MacKey`, used only in the next message, and the `LTK`, which is the shared long-term secret key. f5 is defined such that

$$\texttt{MacKey} = \text{AES-CMAC}_T(0 \,||\, \text{“conf”} \,||\, N_\text{a} \,||\, N_\text{b} \,||\, \text{ID}_\text{a} \,||\, \text{ID}_\text{b})$$
$$\texttt{LTK} = \text{AES-CMAC}_T(1 \,||\, \text{“conf”} \,||\, N_\text{a} \,||\, N_\text{b} \,||\, \text{ID}_\text{a} \,||\, \text{ID}_\text{b})$$

where the 0/1 bit ensures the two keys are different; the fixed string "conf" identifies the purpose of the MAC, preventing it being used in another context if the protocol were ever extended; the two nonces introduce entropy; and the IDs tie the generated keys to the devices' IDs. The protocol defines

$$T = \text{AES-CMAC}_{\texttt{SALT}}(\texttt{DHKey})$$

and `SALT` is the randomly-generated 128-bit hexadecimal constant

```
SALT = 0x9DD4 3105 BA69 0FCF 3E46 BA9C FEE5 ECE9
```

Bluetooth uses a different `SALT`, but changing it here gives the slight benefit that rainbow tables generated for Bluetooth's `SALT` value can't be re-used against TypeSafely.

f5 uses T as a key, rather than `DHKey` directly, because ECDH-derived keys aren't completely random; they represent a point on the curve. This means that some bits of the key may be predictable, which could weaken operations that use the key later. For this reason, it's good practice to use a key-derivation function (KDF) to extract this entropy while making all the bits unpredictable (*Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, Ch. 5.8) [4]. This method, using AES-CMAC with a fixed salt as the key, is one way of doing so [3].

Once these keys have been generated, `MacKey` is used as the key for generating the confirmation E_x values using f6:

$$\text{f6}(\text{K}, b, N_1, N_2, \text{ID}_i, r_i, x_i) = \text{AES-CMAC}_K(b \,||\, N_1 \,||\, N_2 \,||\, \text{ID}_i \,||\, r_i \,||\, x_i)$$

$$E_x = \text{f6}(\text{MacKey}, b, N_x, N_y, \text{ID}_x, r_x, \text{x\_knows\_y})$$

The boolean b distinguishes the host's E_x from the keyboard's. The confirmation values E_x authenticate the un-authenticated parameters sent earlier: tampering with either side's ID, the PIN, or the *x-knows-y* values will give a different E_x here. Bluetooth passes different arguments to f6, since it has different unauthenticated parameters that aren't used in TypeSafely. An attacker can't falsify an E_x value, since they can't know `MacKey`.

If a device can't reproduce the other side's E_x value, it resets the pairing and sends an error message. Otherwise, the pairing is deemed to be successful and both sides store the LTK. In practical implementations, it's a good idea for the host computer to check that the user is authorised to make a new pairing, such as by asking for their login password.

Next, the host will jump to the re-connection stage, described in subsubsection 3.2.1.5.

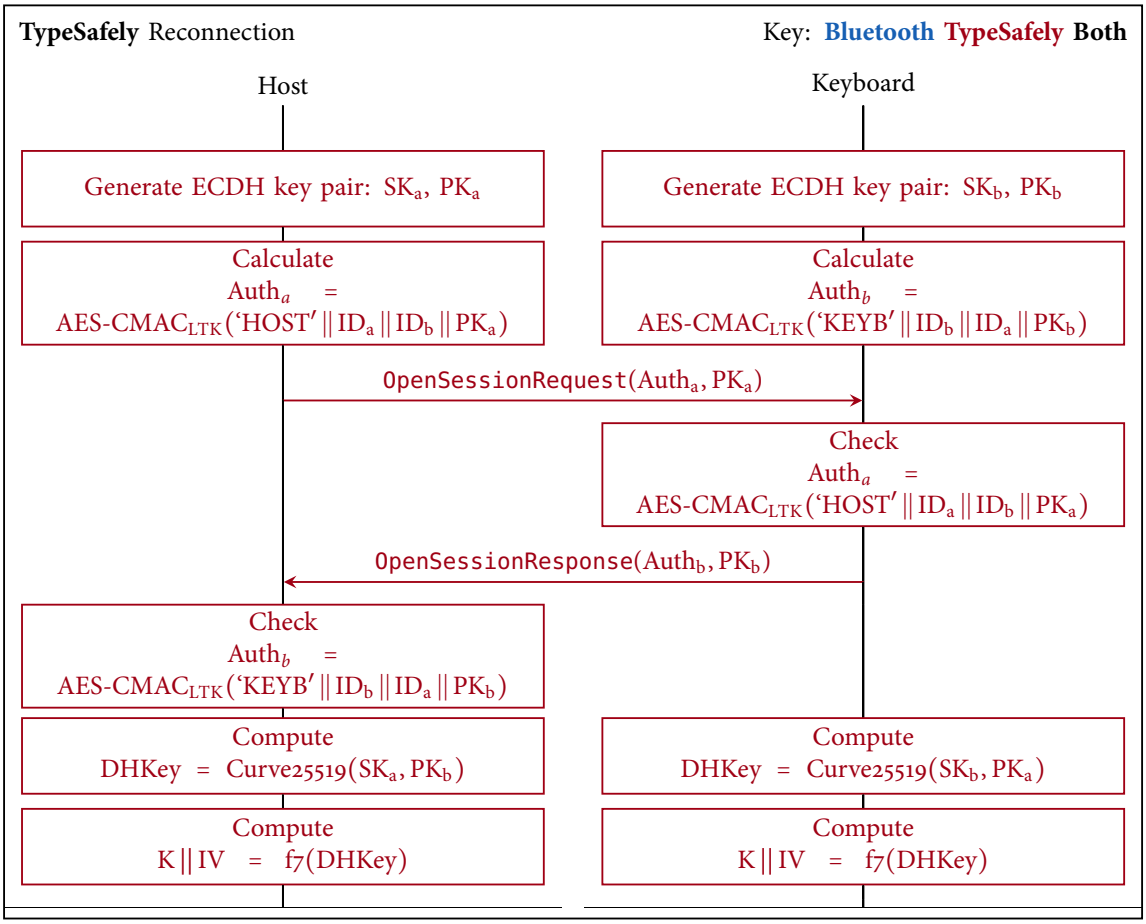


Figure 3.8: Sequence diagram for protocol stage 5 – Reconnection stage

3.2.1.5 *Reconnection stage*

At this stage, both sides believe they know the LTK. This stage is *not* taken from Bluetooth—it isn't part of the pairing process—but shares many similarities with the pairing's key exchange process (see subsubsection 3.2.1.2).

First, the host generates an ECDH keypair SK_a, PK_a. The keypair is only used during the reconnection stage, and is discarded once the session is open. Next, the two calculate *Auth* values. These values are calculated as the MAC, with *LTK* as the key, of the concatenation of:

- The fixed string "HOST"/"KEYB", distinguishing the two devices' *Auth* values. This precludes an attacker, masquerading as the keyboard, returning the host's MAC back to it.
- The two devices' IDs, implicitly authenticating them. Note the *x-knows-y* values aren't sent—there's no need, because they must have both been `true` for this stage to be entered.
- The device's public key, tying the *Auth* value to that particular device.

Since the *Auth* values use *LTK* as a key, they inherently prove that the sender knows *LTK*. Once they have computed this, the host sends an `OpenSessionRequest` containing its *Auth*

value and public key. The keyboard then checks the host's *Auth* value and responds with a similar message. If the check fails, the device sends an error indicating that a new pairing needs to be made.

Once the host has validated the keyboard's *Auth* value, the two sides compute the shared `DHKey` and from this derive a session encryption key and initialisation vector using f7. f7 is defined such that

$$\begin{aligned} \texttt{K} &= \text{AES-CMAC}_{\texttt{SALT}}(0 \,||\, \texttt{DHKey}) \\ \texttt{IV} &= \text{AES-CMAC}_{\texttt{SALT}}(1 \,||\, \texttt{DHKey}) \end{aligned}$$

and `SALT` is arbitrarily chosen to be

$$\texttt{SALT} = \texttt{0x1234 56AC 1D1C 5A17 0068 6267 6A32 0000}$$

An explanation of the security of deriving keys from an ECDH key in this way was discussed in subsubsection 3.2.1.4.

Here, an attacker can't know `K` or `IV` as they are derived from `DHKey`, which is only known by the two devices. Note that because the ephemeral `DHKey` is used to derive the encryption key, rather than the LTK, the protocol guarantees Perfect Forward Secrecy: if an attacker later gets access to `LTK`, they still won't be able to decrypt any past communications.

Now that these values have been calculated, the session is considered to be open—see subsubsection 3.2.1.6.

3.2.1.6 *Connected stage*

Once the session is open, the keyboard sends packets containing the current set of pressed keys every 1 ms—regardless of whether any keys are pressed. This, combined with the use of encryption, precludes an attacker inferring keystrokes based on intervals between keypresses.

AES-CCM is used to encrypt the packets, with key K from the previous stage. AES-CCM is an authenticated encryption scheme, also used by Bluetooth, that gives the encrypted ciphertext C and an authentication tag T for each encryption. Authentication is crucial here, as my threat model means an attacker can control or predict which keys are pressed at certain times. A malleable encryption scheme would therefore allow an attacker to flip bits to change which keys are pressed, despite not knowing the encryption key.

AES-CCM requires a using a unique nonce per encryption round, or an attacker might be able to gain insight into the plaintext. To ensure uniqueness, nonces are defined to be the sequence number XORd with the IV, and the sequence number is incremented in each message. While a secret IV isn't necessary with AES-CCM, I've added it for some marginal extra security: the cost of including it is minimal, and mass-cryptanalysis attacks have recently been theorised [6] against systems without it. Having a monotonous sequence number prevents replay attacks: without this, an attacker could repeat previous keypresses by re-sending those messages.

When the host receives a packet, it decrypts and validates the ciphertext and tag, thereby ensuring the message hasn't been tampered with. If this fails, the message is discarded. It then validates and updates the sequence number–if the received sequence number has already

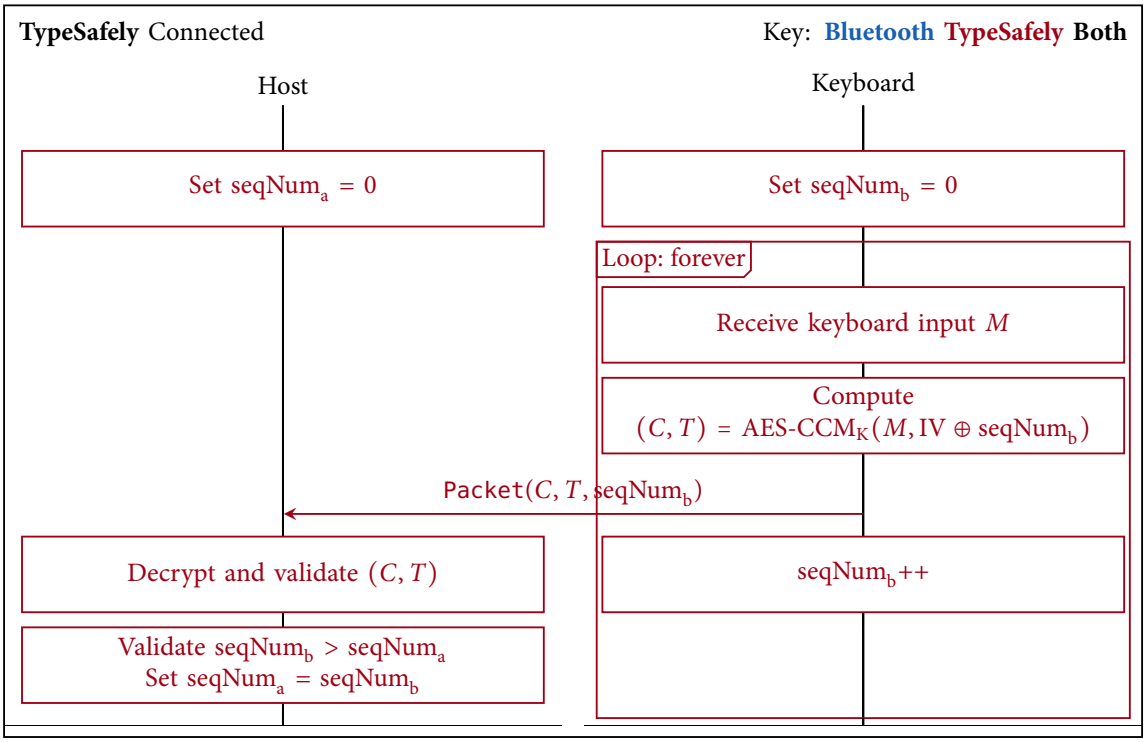


Figure 3.9: Sequence diagram for protocol stage 6 — Connected stage

been seen, then the message is discarded. Finally, it accepts the plaintext as valid keypress data.

3.2.2 WIRE FORMAT

To send messages over-the-wire in a portable manner, they must be serialised. I chose to use Google's Protocol Buffers (Protobuf) format [22] for this. Protobuf is a widely-used lightweight binary format for encoding structured messages. There are Protobuf implementations for many platforms, and—largely due to being a binary format—encoding/decoding messages is very fast, which is helpful for minimising latency.

The schema of a Protobuf message must be known before decoding it, which makes Protobufs less flexible than some alternatives. However, this allows messages to be statically typechecked against the specification, thus increasing safety. It also keeps the encoding compact, which should help messages fit under USB Full Speed's 64-byte packet-size limit.

With the protocol design finished, I created Protobuf files specifying the format and typing for each message. An example of such a file is included in Listing A.1.

3.2.3 PYTHON REFERENCE IMPLEMENTATION

To test and validate the protocol design, I created a reference implementation in Python. A clear reference implementation was useful: being confident in its correctness gave solid test vectors for my 'real' implementation, and helped me work out a good code structure.

I chose Python for this task due to its sizeable standard library, which allowed me to call on many library routines to keep my code terse and readable. Being a reference implementation, performance didn't weigh heavily here.

3.2.3.1 *Structure*

Emphasising simplicity, the Python implementation consists of 3 main files: one for the protocol implementation, a file containing utility functions, and a file that performs test run-throughs of the protocol. There's also some Message classes auto-generated from the Protobuf specifications by Google's `protoc` compiler (see subsection 3.2.2).

The protocol implementations are finite state machines (FSM), implemented as instances of a central `StateMachine` class, which keeps track of the current state and has memory for data such as private keys. The states roughly correspond to the different messages that may be expected at any given time: `AWAIT_INITIAL_CONNECTION`, `AWAIT_PAIRING_AUTHENTICATION`, etc.

State machines are a good fit for this situation: given an input message, the device processes it differently depending on the state, outputs a message, and enters a new state. Using a series of states allows for strong portability—a state-based model can fit right into the event-loop based architecture of most embedded code. This is a huge benefit when designing for microcontrollers.

Explicitly designing as a FSM keeps the implementation simple and robust, mostly because the logic of each state remains entirely separate. This allows for clear definition of pre/post-conditions of the state, and permits validating a single state's behaviour in isolation. Additionally, knowing that the individual states are correct gives confidence that the overall implementation is correct, since there's very little 'glue' code between the states.

```
def state_await_session_upgrade(input_msg: Message,
                                ltk_store: dict,
                                memory: TSProtocolDriverMemory)
                                -> (DeviceState, Optional[Message]):
    [ ... ]
    return DeviceState.SESSION_OPEN, messages_session_pb2.UpgradeResponse()
```

Listing 3.1: Sample state machine function

Each state is implemented as its own function with its own logic for validating the input message, but most simply check that the message is of the expected type and has correct field sizes. The states then perform their processing, and return which state comes next as a value from an enum, and optionally a message to output.

The central `StateMachine` class has a `tick` method. This reads any messages in, calls the current state's function, updates the class's `state` value to the next state, and returns the output message. The method looks up the current state's enum value in a subclass-defined `dict` to find its function pointer.

3.2.3.2 *Implementation*

I incrementally implemented stages as I went, letting me work out any kinks I found in the specification before they became too established. This led to me fixing broken aspects of my initial protocol design, and reducing the number round-trips required for a complete connection.

I used Python-native libraries for the cryptographic primitives (AES and ECDH). These have different APIs to the libraries used in my C++ implementation, which could help detect bugs caused by misusing one of the APIs in a subtle way.

Many of the protocol's functions, such as f4, are based on Bluetooth functions of the same name. To give confidence in my implementation, I first implemented these functions exactly as they appear in Bluetooth, which let me check my implementations against the Bluetooth-provided test vectors. I could then carefully make the minor tweaks to fit my protocol and have confidence that the functions still gave correct results. This turned out to be prudent: my initial implementation of one function misused a parameter, giving incorrect results. This bug would've otherwise been difficult to diagnose.

Despite not being used in my 'real' codebase, it was important to include sensible error handling to exemplify the correct behaviour in such cases. Therefore, assertions and data validation routines are included throughout.

3.2.3.3 *Testing*

As this was a reference implementation, rather than one to be used in my final project, the majority of my testing of this implementation was done by hand. This included checking that incorrect cryptographic confirmations, unexpected messages, and other unexpected behaviour from the other end of the connection would give the result specified in the protocol specification.

3.2.4 C++ IMPLEMENTATION

The C++ implementation of the protocol is used in both the microcontroller firmware and the host daemon. It's based on the reference implementation, but is more portable, robust, and efficient, in order to work as well on a microcontroller as on the host computer.

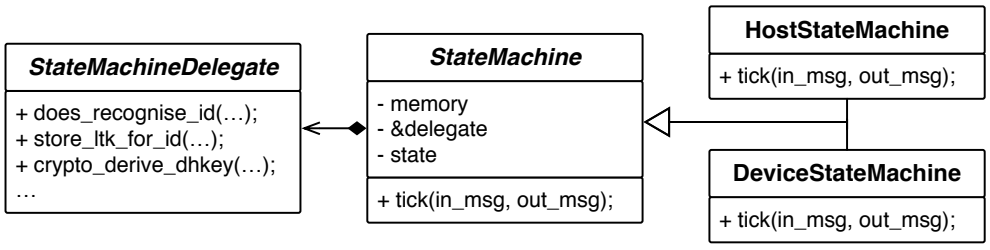


Figure 3.10: UML diagram representing high-level structure of the C++ protocol library

For maximum portability, the C++ implementation is designed to be a library. The library relies on the outside codebase to pass in an instance of the `StateMachineDelegate` class. This class exposes some non-portable implementations needed by the library, such as cryptography, I/O, or event-handling outside the library.

This allows the same library to be easily used on vastly different platforms. For example, I created a small, host-only codebase to test the implementation without having to deploy new versions of the firmware and daemon. This testing codebase provides a minimal, basic implementation of all the delegate methods. For example, it automatically enters the PIN, and only stores pairings in memory.

3.2.4.1 *Structure*

The code is structured similarly to the Python version; there's a virtual `StateMachine` class representing a protocol state machine, and two subclasses: `HostStateMachine` for the host, and `DeviceStateMachine` for the keyboard.

Each `StateMachine` subclass has a function implementing each state, and states are identified by values from an enum. Just as with the Python implementation, the functions receive the input message wrapper and references to the memory and delegate object as input. They return the next state (as a value from an enum) and optionally a message wrapper to output.

3.3 MICROCONTROLLER ABSTRACTION LIBRARY

As detailed in subsection 2.4.1, I chose to create a platform-abstraction library—built in C++ on top of libopencm3 [18]—to provide a single interface for interacting with microcontroller features, giving portability and separating the platform-specific code from the main codebase. While the library is currently only implemented for the STM32F415, structuring my code in this way will simplify porting the code to another platform if needed, and helps keep the main codebase terse.

The abstraction library exposes the following features: GPIO pins, serial ports, timekeeping methods, hardware-accelerated cryptography, a TypeSafely keyboard interface, an EEPROM, and a hardware-accelerated CRC checker. I've documented a few of the features in this section.

3.3.1 GPIOS

The library's GPIO interface is a prime example of the benefits of the platform abstraction the library brings. Crucially, the functions it exports, such as (full header extract given in the appendix: Listing A.2)

```
void WritePin(Pin pin, bool value);
```

operate on a `Pin` type, instances of which can't be created outside the abstraction library. Instead, the library exports `extern const Pin` values that may be passed to these functions. This prevents the main code from being tied to a microcontroller-specific pin layout, since

the platform implementation exposes only pins by name—such as `PinLED`—rather than by pin numbers.

3.3.2 CRYPTOGRAPHY

The process of creating the cryptography implementation also shows the benefits of isolation between the main codebase and the microcontroller-specific code. Cryptographic functions are included in the library interface to allow hardware implementations on some platforms, but my initial implementation only used software libraries.

I used the curve25519-donna [17] library for Curve25519 as it's portable, small, and widely used, and therefore is likely to be bug-free. For AES-CCM and AES-CMAC, I used cifra [10], a library of cryptographic primitives designed for embedded systems. The hardware-abstraction library then just provided a standardised interface in front of these functions, plus a function to get data from the built-in hardware random number generator.

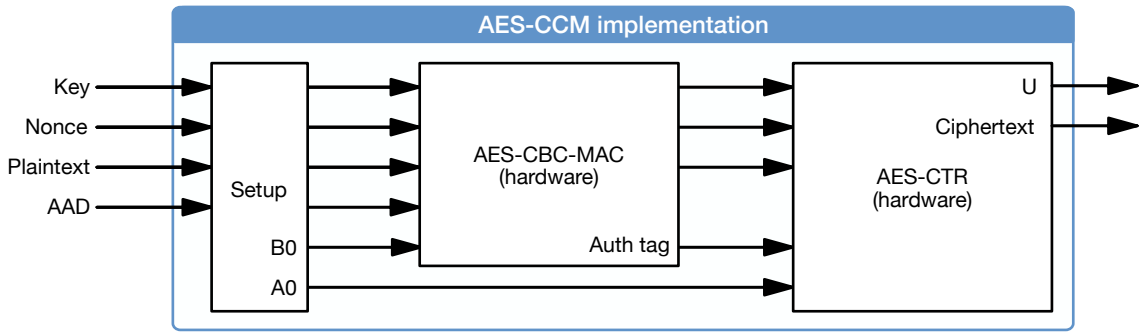
However, while performing my evaluation, I discovered that cifra's AES-CCM implementation introduced an unacceptably high amount of latency (see subsection 4.4.1). Thankfully, I'd chosen my microcontroller for its hardware AES-CCM implementation (see section 3.1). Using the hardware AES-CCM is complicated; all setup work, such as creating the header blocks, has to be done in software—the hardware only accelerates the actual encryption. As such, this process required much studying of the microcontroller datasheet [23] and the AES-CCM RFC [34].

Even after many hours of debugging, my implementation didn't seem to be working: I'd independently stumbled upon a long-running bug[1] in libopencm3: it subtly `#define`'d the size of the memory region for the AES IV incorrectly. Despite patching this bug, my AES-CCM implementation *still* didn't work. It was only then that I discovered STMicoelectronics had mistakenly listed the microcontroller as supporting AES-CCM in some parts of the datasheet[2], and while my code was correct, the microcontroller was performing some other function instead of AES-CCM.

Thankfully, the computationally expensive part of AES-CCM is built on top of the AES-CBC (for authentication) and AES-CTR (for encryption) modes [34]—both of which *are* supported in hardware. Therefore, I was able to use the hardware implementation of these primitives to build my own *mostly* hardware-accelerated CCM implementation. I used the RFC-provided test vectors to check my implementations of the CBC and CTR modes, before checking my overall CBC implementation.

1 Existing pull request here: https://github.com/libopencm3/libopencm3/pull/897

2 Section 23.6.2 - "CRYP control register" of the STM32F4 reference manual (RM0090 Rev.17) [23] lists CCM support for the STM32F415.



Since this method requires the CBC and CTR rounds to be run sequentially, this would be slower than a full hardware implementation, but I estimated that this would still be much faster than the software implementation. This turned out to be correct, giving a 500x speedup (see subsection 4.4.1). Having a platform-abstraction library allowed me to make and test these changes in isolation, which would have been otherwise difficult.

3.3.3 USB

This part of the library exists to expose methods to send and receive data from both the standard USB boot keyboard interface and the TypeSafely interface, but under-the-hood it also handles setting up and controlling the microcontroller's USB core.

The logic of responding to USB control requests with the USB device descriptors, queueing data to be transmitted at the correct time for maximum throughput/minimum latency, and handling (dis-)connections is all hidden from the main codebase, which aids readability.

Notably, the USB system also has to record the time queued packets are actually read by the host. It detects possible host failures using an interrupt handler that resets a timer when packets are actually read. If the timer expires, it assumes the host has disconnected, and calls a callback provided by the main codebase. In my implementation, the keyboard would then revert to the standard insecure Boot Keyboard mode—it might indicate that the user has rebooted into their BIOS, for example.

3.4 KEYBOARD FIRMWARE

The keyboard firmware is the main codebase of TypeSafely's microcontroller implementation.

Most of the execution happens inside the `KeyboardStateMachine` class. This class contains a `DeviceStateMachine` instance, and acts as its delegate (and therefore implements methods such as `begin_passkey_entry` for it—see subsection 3.2.4).

3.4.1 THE KEYBOARD STATE MACHINE

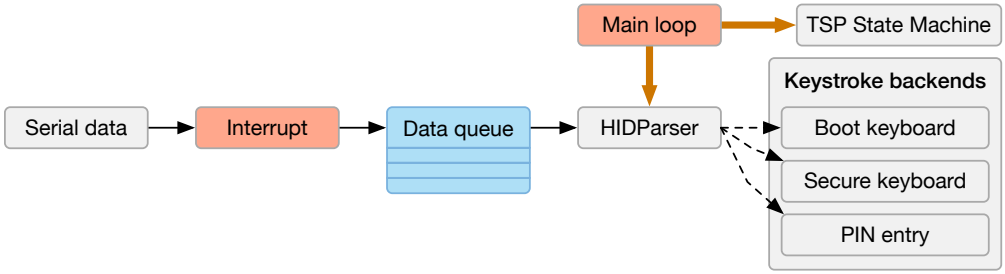


Figure 3.11: Diagram showing both control flow and progression of keystroke data through the `KeyboardStateMachine` class

The `KeyboardStateMachine` has a `tick` method that gets called in a loop by the firmware's `main` function. This `tick` method calls both the `HIDParser` and the `DeviceStateMachine` to perform their processing. Rather than polling in a loop, an interrupt triggers reading in new keypress data over serial when it becomes available. The interrupt handler stores this data in a custom-made circular buffer, so that it can later be read out by the `HIDParser`. Since these interrupts can happen in the middle of execution, this custom circular buffer implementation has to be designed to be 'interrupt-safe'—analogous to thread safety. Mutual exclusion is achieved by disabling interrupts during critical regions, delaying any interrupts that are triggered during execution.

The `HIDParser` reads ASCII-encoded hexadecimal USB HID report data from a queue until a complete set of keystrokes have been obtained. It then decodes the hexadecimal-encoded data into raw bytes, which it passes to the current keystroke backend. The `KeyboardStateMachine` keeps track of which is the current backend. These backends then process the keystrokes: the Boot keyboard backend forwards them to the host unencrypted, the Secure keyboard backend encrypts them first, and the PIN entry backend parses them to a number.

3.4.2 PERSISTENT STORAGE

To permit re-connection across reboots, the `KeyboardStateMachine` requires a non-volatile way to store the keyboard's ID and its pairings (as ID-to-LTK mappings).

For this, I created a `PersistentLTKStorage` class which tries to load these values from EEPROM when the microcontroller is first powered up, caches the results, and writes updated data back to EEPROM.

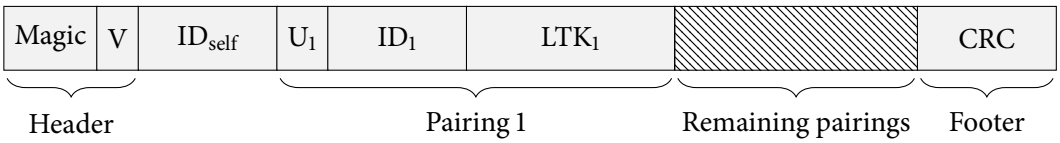


Figure 3.12: Diagram showing format of data written to EEPROM.

Writing to EEPROM was designed to be robust, since a corrupted LTK could raise a false connection-tampering alarm. The data is written to the EEPROM at a fixed address, and formatted as a header and a footer around the keyboard's own ID and a fixed number of contiguous entries. Each entry includes a '*U*' bit indicating if it's used, the host's ID, and the LTK. The header has two fields: a 2-octet 'magic' number, allowing `PersistentLTKStorage` to identify if the data it reads is initialised or not, and a version number byte *V*, allowing extensibility in the future. The footer is the CRC32 of all previous data, which allows the class to check if the data is corrupted when it's read back.

3.5 DAEMON

The Linux daemon is the host portion of the TypeSafely system. Like the microcontroller firmware, it's written in C++ and makes use of the core protocol library. To simplify setup, it currently must be run as `root`, but is designed to be capable of running under its own user with limited privileges. This would reduce the potential impact of vulnerabilities in the code: a bug in the daemon couldn't give an attacker full, root-level access to the computer.

3.5.1 STRUCTURE

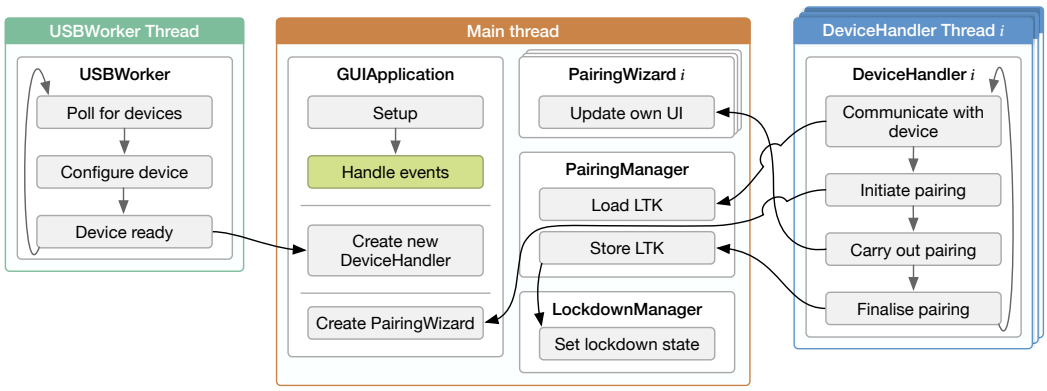


Figure 3.13: Diagram showing the actions and communications between objects on different threads. Each box inside a thread represents an object, and the grey boxes inside objects roughly correspond to different methods. Black arrows indicate one method calling another.

To reduce complexity, the daemon is responsible for the GUI, rather than having a separate GUI process running under the user's account. Therefore, the main thread shouldn't perform actions that might block, or the GUI will appear to be unresponsive.

The daemon should be able to interact with many TypeSafely keyboards simultaneously, so interaction with each device occurs on its own `DeviceHandler` thread. This allows blocking IO to be used, since blocking on one device won't affect the others, which greatly simplifies the implementation. Additionally, the detection and initial connection to new USB keyboards is handled by a separate `USBWorker` thread. Finally, the main thread coordinates communication between threads, handles the GUI elements, manages the set of paired devices, controls the keyboard-blocking ('lockdown') status.

Inter-thread communication is done using Qt's signal/slot mechanism, which allows objects to safely call methods on objects on other threads. Since Qt requires UI activity to be done on the main thread, this mechanism is taken advantage of to communicate events (such as `PinEntryComplete`) between the `DeviceHandler` threads and their accompanying `PairingWizards`.

AES-CCM and AES-CMAC are implemented using OpenSSL's `libcrypto` library. However, some distros use older `libcrypto` versions lacking Curve25519 support, so for portability this is implemented using `Curve25519_donna`—like on the microcontroller.

Settings and pairings are stored in the `/etc/typesafely` directory. The directory is automatically created with `0700` permissions, preventing unprivileged users from accessing any stored secrets.

The cross-platform `libusb` framework is used for communication over USB. On its own thread, the daemon sets up a callback for when a device matching the TypeSafely keyboard's USB vendor/product ID is connected. This callback configures the device to the correct mode, and passes the resulting `libusb` device handle back to the main thread. The main thread spawns a `DeviceHandler` thread, which performs all further communication with that device.

3.5.2 KEYBOARD CONNECTION FLOW

The UI runs on the main thread, and uses the Qt framework to manage the GUI. When a new keyboard is detected by the USB worker thread, it notifies the main thread, which then creates a new device-handler thread to communicate with the device. Each device-handler thread contains its own `HostStateMachine` (documented in subsection 3.2.4), allowing many different keyboards to be independently used at once. The device-handler threads act as a delegate for their state machines, but occasionally have to call out to the main thread to update the GUI (e.g., for the `PairingInitiated` or `PinEntryComplete` events).

The UI was designed to be clear and unintrusive. If the device-handler finds no existing pairings, it immediately initiates pairing. It notifies the main thread, which creates a new `PairingWizard` and connects its signals/slots to the device-handler. The `PairingWizard` manages all UI of the pairing procedure, and emits events (such as page changes) back to the device-handler directly. Thanks to the structure of the program, this inter-thread communication is as simple as this:

```
emit gui_worker->displayPin(pass_num); // Display calculated PIN onscreen
```

The `PairingWizard` displays the window shown in Figure 3.14, which lets the user know how to pair their keyboard. Clicking 'Next' displays the PIN. Once this PIN is entered, the assistant informs the user that pairing has succeeded, and exits. If the device-handler discovers an already-paired keyboard, it automatically and invisibly connects to it, as if it were a standard keyboard.

For clarity, if the user's desktop environment supports desktop notifications [15], the daemon sends a descriptive notification when it blocks a newly-connected keyboard. This is shown in Figure 3.15.

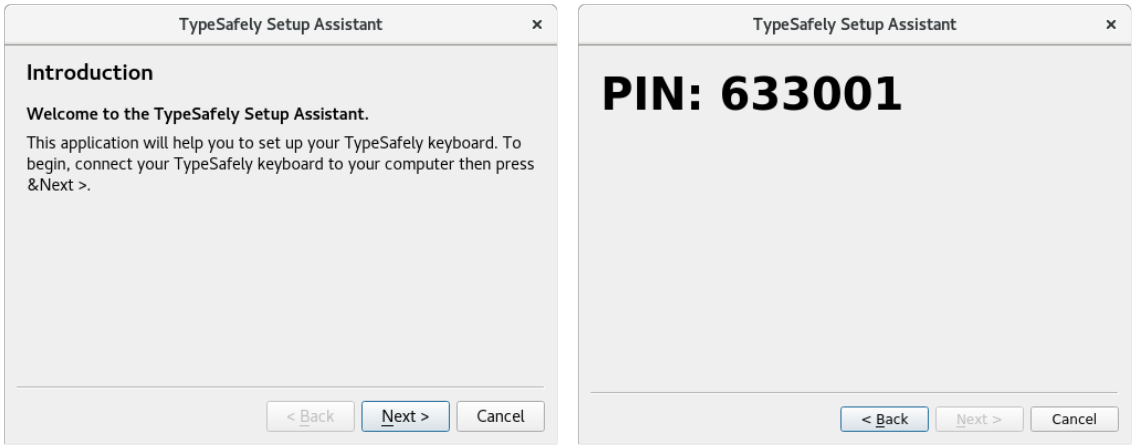


Figure 3.14: Screenshots of the setup assistant.

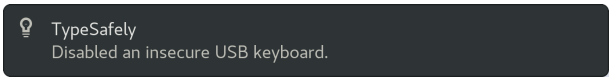


Figure 3.15: Screenshot of notification delivered when a insecure keyboard is disabled on connection

3.5.3 HANDLING KEYBOARD INPUT

Once a device-handler has an open session with its keyboard, it creates a *virtual* keyboard using Linux's `uinput` module. The code for this is based on the documentation's example [30] that simply presses the space bar. I adapted this example to support full keyboard input.

The thread reads encrypted keystroke data from the keyboard in a loop, polling with a long timeout. Once it reads new data, it decrypts it and passes it to `uinput`.

3.5.4 BLOCKING INSECURE KEYBOARDS

Insecure keyboards are blocked using an automatically-installed `udev` rule, which runs for newly-connected USB device interfaces that use the USB keyboard protocol—which in effect limits the rule to newly connected standard USB keyboard interfaces. Notably, this will only disable the *keyboard* interface: if a device presents other interfaces, such as in a combined keyboard-and-mouse, those won't be disabled.

The rule then runs a Bash script, which reads the file at

`/var/run/typesafely/lockdown_enabled`

If the file exists and its contents is '`1`', the script will disable the keyboard interface by running

```
echo 0 > /sys/$DEVPATH/authorized
```

The `lockdown_enabled` file is in the `/var/run` tempfs, so will be erased on reboot. The daemon writes a 1 or a 0 to the file to disable/enable insecure keyboards. By default, 1 is written so long as any secure keyboards are paired. This behaviour comes from the desire

for TypeSafely to be invisible by default, rather than immediately disabling any insecure keyboards when the program is first installed.

After updating the `lockdown_enabled` file, the daemon will also execute `udevadm trigger`, which forces `udev` to re-run all rules on currently connected devices. This has the effect of disabling (or re-enabling) any currently-connected insecure keyboards.

3.6 REPOSITORY OVERVIEW

Below is a high-level overview of the repository structure. Table 3.1 describes all non-library code not wholly written by myself. A more detailed, unabridged overview is given in the appendix: section A.1

```
├── README.md - Informational file for the repository (44)
├── common - Code used by both the microcontroller firmware and the host (2187)
│   ├── protobuf - Contains the Protocol Buffer files, specifying the protocol (106)
│   ├── protobuf_test_cpp - C++ program: tests the C++ protocol impl. (229)
│   ├── protobuf_test_python - The Python reference protocol implementation. (826)
│   └── protocol_state_machine - The C++ protocol library (1606)
│       ├── messages*.[c, h] - nanopb-generated protocol structures
│       ├── pb*.[c, h] - nanopb library files
│       └── tsp*.[cpp, hpp] - Core protocol library code
├── type-safely-firmware - Codebase for the keyboard firmware (2581)
│   ├── libopencm3 - libopencm3 library
│   ├── platforms - Platform-specific implementations live in here (1426)
│   │   └── STM32F4 (1426)
│   │       ├── external - 3rd party code
│   │       │   ├── cifra - Cifra cryptography library
│   │       │   └── curve25519_donna - Curve25519_donna library
│   │       └── src - Platform abstraction library code (1426)
│   └── src - Main firmware codebase (1062)
└── type-safely-daemon - Host daemon codebase (1614)

(Line counts for directories shown in brackets. Excludes comments, blank lines,
auto-generated code, and libraries.)
```

File	Description
daemon: usb.c	~81 LoC to fetch USB endpoint addresses and configuration descriptors from a libusb device handle was derived from my work on another project
firmware: PlatformEEPROM.c	The EEPROM interaction code was largely based on eeprom_driver, Copyright (c) 2015 Marco Russi. MIT Licenced.
daemon: HIDHandler.cpp	`uinput` code to send keystrokes to the Linux kernel was derived from the documentation's example code (see subsection 3.5.3 for details).

Table 3.1: Table describing all derived code in the project. (N.B.: Doesn't include libraries, which are detailed in the above directory tree.)

Evaluation

4

In this chapter, I describe the steps I took in order to evaluate the performance and functionality of my project. I first describe how I verified the project was working as expected and I assessed its effectiveness against realistic threats, before finally appraising the efficiency by measuring timings and resource consumption.

4.1 Implementation Verification

Verification is especially important in any security-related project—even the subtlest of slip-ups can cause catastrophic security failure. As such, I was sure to scrutinise that my implementation, to ensure it matched the design.

4.1.1 Protocol Implementation

To give confidence in my final implementation, I first created a reference implementation in Python (detailed in subsection 3.2.3). The reference implementation was designed for clarity, and for this reason has variable names that match the protocol specification.

Many helper functions, such as f4, are based on Bluetooth-specified functions. I first implemented these functions exactly as they appear in Bluetooth, which let me check my implementations against the Bluetooth-provided test vectors. I then carefully made the minor tweaks necessary to fit my protocol, which gave confidence that the functions were still correct. This turned out to be prudent: my initial implementation of one function misused a parameter, giving incorrect results.

I also manually stepped through the implementation's connection process to ensure that the code's operation matched my expectations. Because each state is implemented by a single function, I could asses the states largely in isolation, which reduced the effort required to verify the entire system.

Next, I created the C++ protocol library (described in subsection 3.2.4). This code is a close analogue of my Python implementation, allowing me to track down any bugs by comparing the state of the two at a given point with the same inputs. Additionally, the two use a common message encoding (see subsection 3.2.2), which allowed me to verify that the two were performing compatible computations.

By the nature of the project, to an extent each device inherently validates the other's implementation: if one side has an incorrect state or cryptography logic, it's likely that this will cause a disagreement in a computed check value or some other mismatch that would result in an error. I also checked numerous test cases against my Python implementation, and added some automated tests to check the implementation end-to-end.

Finally, I checked that the data I was expecting to be sent was actually sent over-the-wire by using a logic analyser. This multi-layered approach gave me confidence that my implementation was actually following the specification correctly.

4.1.2 KEYPRESS SPOOFING PROTECTION

TypeSafely disables non-secure keyboard input whilst a secure keyboard is paired. This prevents harm when an attacker forces a TypeSafely keyboard into insecure mode, or connects their own keyboard to send keystrokes.

I verified that the 'lockdown' mechanism was inactive when no TypeSafely keyboard was paired.

I then paired a TypeSafely keyboard and re-ran the test with three different standard keyboards I had to hand (branded Dell, Microsoft, and Rii respectively). All three were successfully blocked, with the popup shown in Figure 4.1.

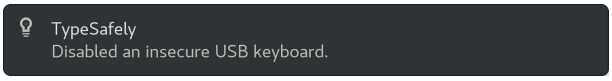


Figure 4.1: Screenshot of the notification that appears once an insecure keyboard is connected. The appearance of the notification depends on the user's desktop environment.

The blocking mechanism only detects standard USB keyboard interfaces. As a result, I suspect that some specialist gaming keyboards that use their own drivers (to enable features like n-key rollover) might actually bypass this block. This is a caveat that users should be warned about.

4.1.3 IMPLEMENTATION FRICTION AND CAVEATS

Due to time constraints I was unable to carry out the 'usability study' project extension to methodically assess the friction of the system. However, I can say objectively that:

1. Once set up the system behaves exactly as a standard keyboard would, with the caveat that the user should avoid typing into a keyboard that's flashing to indicate an insecure connection (see section 3.1).
2. The pairing process only needs to happen once, and the automatically-appearing UI explains clearly explains to the user the steps; all they need to do is enter the 6-digit PIN shown onscreen into their keyboard.

Given the above, I hypothesise that most technically-literate users would not have a problem with using the system. However, it would be worth conducting a user study to assess what could be improved.

One caveat is that due to the daemon's design, TypeSafely can only be used once the OS is loaded. This means that keypresses to any pre-boot environment, such as a full-disk encryption password entry screen, will be unsecured. This could be overcome using an additional host module for pre-boot environments such as EFI, however this would require much development effort.

4.2 EFFECTIVENESS AGAINST MODELLED THREATS

It's important to evaluate how well the system protects against attacks that are possible within the bounds of the threat model. In order to convincingly and comprehensively explore the possible threats, I used the standard STRIDE model[26, Ch. 3], a mnemonic representing each different category of threat that might be considered in a secure system. The following subsections detail how TypeSafely prevents the USB keyboard attack(s) made possible by each threat.

STRIDE threat	Property violated	TypeSafety attack class
Spoofing of user identity	Authenticity	Keyboard spoofing: subsection 4.2.3
Tampering with data	Integrity	Keypress modification: subsection 4.2.4
Repudiability	Non-repudiation	Not necessary in this system: hosts don't mind which keyboard sent keypresses, so long as they're legitimate
Information disclosure	Confidentiality	Passive keylogging: subsection 4.2.1
Denial of service	Availability	Not considered (see section 2.2)
Elevation of privilege	Authorisation	Active keylogging: subsection 4.2.2

Table 4.1: Table showing the specific USB-keyboard attack classes for each STRIDE threat. The first two columns are taken from Shostack's *Threat Modeling: Designing for Security* [26, Table 3-1].

4.2.1 PASSIVE KEYLOGGING

Passive keylogging is where an attacker observes keystrokes by eavesdropping on a connection (i.e., without modifying any communications). Here, it's protected against by encryption: with each packet being securely (IND-CCA2 security) encrypted, the attacker can gain no insight into the keys pressed.

However, an eavesdropping attacker could gain insight into keypresses in other ways. For example, in an early version of the project, new packets would be sent whenever a key was pressed or released. An attacker could have used statistical techniques to estimate which keys were pressed, based on the time taken to press them. To resolve this, instead the set of currently-pressed keys is sent every 1 ms—regardless of whether any keys are pressed. This, combined with the use of encryption, precludes an attacker inferring keystrokes based on intervals between keypresses.

Another possible attack arises if the attacker manages to get access to the user's private key at some point, and attempts to decrypt keystrokes they've previously recorded. This is protected against by the use of Perfect Forward Secrecy—see subsubsection 3.2.1.5.

4.2.2 ACTIVE KEYLOGGING

Active keylogging is where an attacker is able to observe keypresses by inserting themselves in-between the host and keyboard and altering connection parameters, without changing any keypresses.

A typical example is where an attacker purports to be the keyboard to the host, and the host to the keyboard. They then decrypt and re-encrypt any communications between the two. This is prevented in TypeSafely's reconnection stage (subsubsection 3.2.1.5) as the *Auth* values validate that both sides know the shared secret LTK, which was agreed during pairing (subsubsection 3.2.1.4).

To get around this, the attacker may try and make their own pairing with the devices. With physical access, they can make their own pairing with the keyboard. However, unless they can read the PIN from the screen whilst the user is logged in (and therefore able to perform a pairing), they can't pair with the host. Since the attacker then presents to the host as an unpaired keyboard, the host will block any keyboard input and alert the user of the attack.

Brute-force-guessing the PIN would be ineffective: there's a 1-in-1,000,000 chance of successfully guessing the PIN in each attempt, and after three attempts a delay is introduced between guesses. The attacker also can't subvert the pairing process by replacing parameters (e.g., a public key) with their own: the commitment values are tied to all parameters previously sent in the pairing, so any attempt to change the values will be rejected.

It's possible for a MITM to initiate pairing with the host while keeping the keyboard in a regular key-input mode. To stop a user being tricked into then entering the PIN onto their keyboard, the UI warns them to only enter their PIN if the keyboard's LED indicates PIN entry mode is active. In this mode, it's impossible for an attacker to learn the PIN from the keyboard (see subsubsection 3.2.1.3).

A downgrade attack is possible: an attacker could modify the USB descriptor packets to hide the TypeSafely interface, so the OS only sees a standard keyboard. However, while a secure keyboard is paired the host would block the keyboard, and so the user would quickly detect this interference. In addition, the keyboard would warn the user that it's not encrypting keystrokes by flashing its LED to attract attention (section 3.1).

4.2.3 KEYBOARD SPOOFING

Keyboard spoofing is where an attacker sends keystrokes by connecting their own device. These attacks are prevented by TypeSafely's authorisation and keyboard-blocking mechanisms (see subsection 4.1.2). As detailed above, an attacker can't falsely pair themselves with the host in order to bypass these mechanisms. The attacker also can't disable this mechanism, as root access to the host is required to do so.

4.2.4 KEYPRESS MODIFICATION

Keypress modification attacks are where an attacker changes the keypress packets sent by a legitimate keyboard. The TypeSafely protocol prevents this by checking the integrity of all keypress data using the AES-CCM *authenticated* encryption mode (see subsubsection 3.2.1.6), and rejecting any packets that have been tampered with.

Replay attacks are a similar attack, where an attacker re-transmits past packets to the host to get certain keys to be re-pressed. This is prevented by the use of sequence numbers, which ensure that if a packet gets retransmitted, the host will reject it. The sequence numbers are also authenticated with the keypress data, and so cannot be tampered with.

4.2.5 ANALYSIS

Within the bounds of the threat model, the above sections describe that the system appears to operate securely. However, in some areas it relies on the user taking the right action (e.g., only entering their PIN when the keyboard is visibly in PIN entry mode). The user is reminded of this in the UI, but ideally the system would be tolerant of users doing the wrong thing.

Additionally, if an attacker is able to observe the onscreen PIN while the user is performing a legitimate pairing, and has a MITM device active, they could provide the PIN to their MITM device and successfully inject themselves into the pairing. This important caveat should be highlighted to users, but a quick look over their shoulder should suffice.

Furthermore, it's certainly possible that there are bugs or flaws in the protocol that haven't been spotted. The security hinges on small details, and a small flaw could derail the security of the whole system—it's happened elsewhere [9]. I'm not a professional cryptographer, and until it's peer-reviewed I wouldn't be comfortable offering this system commercially.

4.3 THREAT MODEL CRITIQUE

The threat model for the project is detailed in section 2.2. Without a realistic and fair threat model, one cannot assess the viability and effectiveness of the project in the real world.

I believe the threat model—a security-weary organisation protecting against an adversary with physical access—to be realistic, and not be contorted to fit what is actually possible. As detailed in the previous section, the project should be secure within these bounds.

However, as stated in the threat model, attacks via tampering with keyboard internals, taking advantage of RF/power usage side channels, and cryptographic timing attacks all may be possible in some cases, but I placed them outside my threat model—very few organisations face (or try to protect against) such threats. However, were this to be released as a real-world product, these attacks would all need careful consideration.

4.4 PERFORMANCE

One of the key measures of the real-world feasibility of this system is the performance overhead versus a standard keyboard. Excessive keystroke latency could lead to an unpleasant typing experience, and non-negligible CPU or memory utilisation could be unacceptable in some scenarios. For this reason, it was important to design the system to minimise these overheads, and to measure the impact quantitatively. The following factors were considered:

4.4.1 KEYPRESS LATENCY

Having a low end-to-end keypress latency is crucial. I initially determined that the system introduced a non-negligible amount of latency using a crude but convenient experimental setup (detailed in section A.2). Based on these rough results, I decided to investigate further.

In order to accurately measure the latency introduced by my hardware and the additional latency introduced by the protocol, I used a more complex setup. This consisted of:

- a Raspberry Pi,
- a Teensy 3.2 [27], connected to the Pi as a USB keyboard that would press the space bar whenever a certain GPIO input is high,
- a cable connecting one of the Pi's GPIO outputs to one of the Teensy's GPIO inputs

The Pi runs a Python script that sets the GPIO high and measures the delay before it sees the spacebar being pressed. The program then releases the spacebar, waits an additional 0.1 s–0.2 s[1], and repeats until n readings have been taken. This setup avoids any latency introduced by the GUI system, and makes accurate, automated measurement of the latency possible.

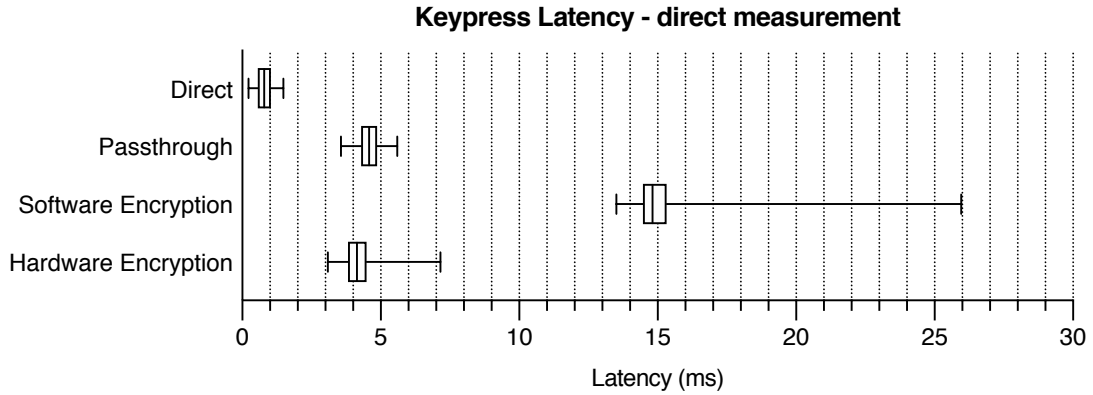


Figure 4.2: Box-and-whisker plot showing the time taken for a keypress to be registered by a userland application running on Linux via a variety of input methods. Lower is better. Whiskers extend to maximum/minimum values. n = 1000 for each method.

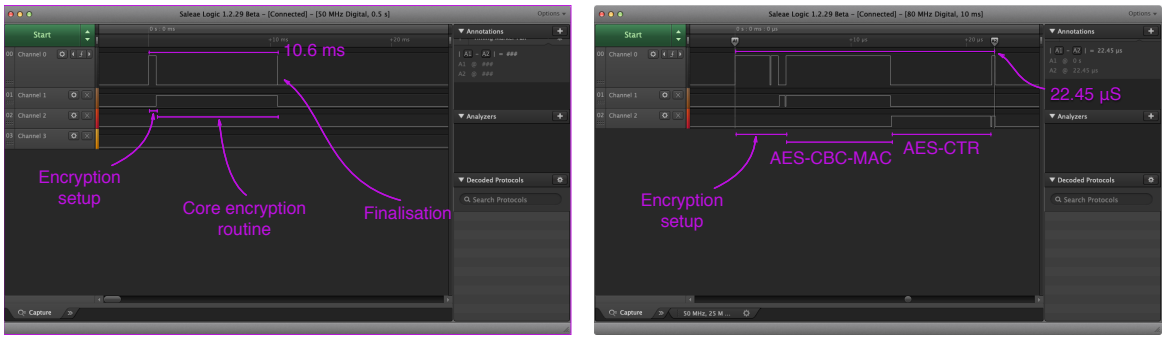
Figure 4.2 shows the results. The 'Direct' result shows a latency of about 0.8 ms with the keyboard connected directly. The 'Passthrough' measurements have TypeSafely inserted

1 Uniformly selected, preventing readings being taken at a fixed phase difference from the underlying USB poll rate

in-between, but only acting as a standard USB keyboard. This shows an average latency of about 4.5 ms—an increase of 3.7 ms, even without any cryptography being performed. There are many different potential sources for this latency, including the additional 'layer' of polling introduced by having a second USB host in between. However, since many keyboards have latencies an order of magnitude greater than this increase [19] anyway, I relatively was satisfied with this figure.

I then measured the latency with the all-software cryptography enabled, which showed a dramatic latency increase of about 10 ms compared to the simple passthrough. This was a more concerning latency figure, since it would often introduce a potentially-noticeable extra frame of latency when displayed onscreen.

To ascertain where the additional latency came from, I used a tried-and-tested technique of embedded performance measurement: I added code to set a unique set of GPIO pins high for each section of the code that I suspected could be a source of the delay. I then used a logic analyser to record the state of the GPIO pins over the course of one keystroke, taking samples at ≥ 50 MHz to get an accurate measurement how long each section of code took.



(a) Software implementation (b) Hardware implementation

Figure 4.3: Screenshots of logic analyser output, showing the execution time of my the microcontroller cryptography implementations. Annotations in pink.

As Figure 4.3 shows, the core encryption routine took nearly 10 ms when implemented in software. Given that this was a single, large target for optimisation, I decided to try a hardware implementation (see subsection 3.3.2 for details).

Figure 4.3 also shows that the hardware implementation's encryption ended up taking just 22.5 µs, compared to 10.6 ms for the software implementation. The large speedup made the difference in latency between the passthrough-only and the fully-encrypted setups negligible (shown in Figure 4.2). After this change, I was satisfied that the latency was sufficiently so low as to not have a negative impact.

4.4.2 PAIRING TIME

Empirically, the processing time during pairing appeared to be very small (less than 1 s), so this metric was not as much of a concern as I thought it would be when preparing my evaluation criteria. Nevertheless, I took 10 measurements of the time taken between initial

connection and pairing completion by logging timestamps during pairing. I excluded the time taken for the user to enter the 6-digit PIN by altering the code to automatically use a fixed PIN for this test.

None of the measurements took more than 500 ms, with an average time of approximately 400 ms. Since pairing happens very rarely, I was satisfied that this figure was sufficiently small to not cause any annoyance.

4.4.3 CPU AND MEMORY USAGE

For these tests, I measured the CPU and memory usage at regular intervals over a 30 minute period. For the tests, I used a 2014 ThinkPad X1 Carbon, which has an Intel i5-4300U CPU running at 1.9 GHz and 8 GB of RAM. This model should fairly represent what a typical user might be using.

In all measurements, memory usage was approximately 25 MB and therefore should be small enough not to be an issue.

With no devices connected, the CPU usage was negligible (displayed as 0%). However, with just one device connected, the CPU usage was consistently about 1.1% (of a single core). Upon investigation, this turned out to be equal parts attributable to USB stack overhead, message decoding, and the actual `HostStateMachine`'s work (e.g., decryption, reporting keypresses to the kernel). All of these things operate 1:1 with each device, meaning that I expect the CPU usage to grow proportionally with the number of connected keyboards. However, I do not expect this to have too much of a negative impact, given how rare it would be for a system to have more than one keyboard connected at once.

5 Conclusions

In the end, this project met all the success criteria outlined in the project proposal.

LESSONS LEARNED

Despite using an existing protocol as a reference, I found designing a secure cryptographic protocol to be even harder than I expected. This was reflected in the number of revisions my protocol went through to eliminate subtle attacks that I'd overlooked. Working on the project further boosted my respect for those who can professionally and convincingly design secure cryptosystems.

Much of the difficulty in developing the project came from writing code that interacts with the microcontroller's hardware. Few embedded developers rely on open-source Hardware Abstraction Libraries and the vendor's documentation alone. As a result, not only is some of the documentation and code available either sparse or incorrect, but it's difficult to find solutions to such problems online. Thanks to the open source community, this is improving over time, but for now extra time should be budgeted for such issues.

SUMMARY

In this project, I successfully created an end-to-end system that protects against keylogging and key injection attacks. This required work in many different areas: security protocol design and implementation; hardware design; embedded, graphical, and device driver development; and post-implementation security analysis.

While there has been work to prevent similar attacks (see section 1.2), this project's strength is that it prevents both keylogging and key injection attacks, even with when faced with a MITM attacker. Furthermore, it includes a usable pairing mechanism, and requires relatively little modification of existing systems to function. As a result, the project provides a method for securing USB keyboards that could realistically be deployed to real users.

Bibliography

[1] Sebastian Angel, Riad S. Wahby, Max Howald, Joshua B. Leners, Michael Spilo, Zhen Sun, Andrew J. Blumberg and Michael Walfish. 'Defending Against Malicious Peripherals with Cinch'. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 397–414. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel.

[2] Chad Austin. *Is It Snappy?* URL: http://isitsnappy.com/.

[3] Elaine Barker, Lily Chen and Rich Davies. *Recommendation for Key-Derivation Methods in Key-Establishment Schemes (Revision 1)*. National Institute of Standards and Technology, 2018. DOI: 10.6028/NIST.SP.800-56Cr1.

[4] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev and Richard Davies. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*. Revision 3. National Institute of Standards and Technology, 2018. DOI: 10.6028/NIST.SP.800-56Ar3.

[5] J. Barnickel, J. Wang and U. Meyer. 'Implementing an Attack on Bluetooth 2.1+ Secure Simple Pairing in Passkey Entry Mode'. In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. June 2012, pp. 17–24. DOI: 10.1109/TrustCom.2012.182.

[6] Mihir Bellare and Björn Tackmann. 'The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3'. In: *Advances in Cryptology – CRYPTO 2016*. Ed. by Matthew Robshaw and Jonathan Katz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 247–276. ISBN: 978-3-662-53018-4. DOI: 10.1007/978-3-662-53018-4_10.

[7] Daniel J. Bernstein. 'Curve25519: New Diffie-Hellman Speed Records'. In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias and Tal Malkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. DOI: 10.1007/11745853_14.

[8] Daniel J. Bernstein, Tanja Lange and Ruben Niederhagen. 'Dual EC: A Standardized Back Door'. In: *The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 256–281. ISBN: 978-3-662-49301-4. DOI: 10.1007/978-3-662-49301-4_17.

[9] Eli Biham and Lior Neumann. *Breaking the Bluetooth Pairing–Fixed Coordinate Invalid Curve Attack*. 2018.

[10] Joseph Birr-Pixton. *Cifra - A collection of cryptographic primitives targeted at embedded use*. URL: https://github.com/ctz/cifra.

[11] *Bluetooth Core Specification*. Version 5.0. Bluetooth SIG.

[12] *Device Class Definition for Human Interface Devices*. Version 1.11. USB Implementers' Forum. June 2001.

[13] Piotr Esden-Tempski. *1Bitsy*. URL: https://1bitsy.org/.

[14] Sean Gallagher. 'Playing NSA, hardware hackers build USB cable that can attack'. In: *Ars Technica* (Jan. 2015). URL: https://arstechnica.com/information-technology/2015/01/playing-nsa-hardware-hackers-build-usb-cable-that-can-attack/.

[15] Mike Hearn, Christian Hammond and William Jon McCann. *Desktop Notifications Specification*. Version 1.2. URL: https://developer.gnome.org/notification-spec/.

[16] HobbyTronics. *HobbyTronics USB Host Controller Board V2.4*. USB Keyboard configuration. URL: https://www.hobbytronics.co.uk/usb-host-board-v2.

[17] Adam Langley. *curve25519-donna*. URL: https://github.com/agl/curve25519-donna.

[18] *libopencm3*. URL: https://libopencm3.org/.

[19] Dan Luu. *Keyboard latency*. Oct. 2017. URL: https://danluu.com/keyboard-latency/.

[20] Matthias Neugschwandtner, Anton Beitler and Anil Kurmus. 'A Transparent Defense Against USB Eavesdropping Attacks'. In: *Proceedings of the 9th European Workshop on System Security*. EuroSec '16. London, United Kingdom: ACM, 2016, 6:1–6:6. ISBN: 978-1-4503-4295-7. DOI: 10.1145/2905760.2905765.

[21] *O.MG Cable*. URL: https://mg.lol/blog/omg-cable/.

[22] *Protocol Buffers*. Google. URL: https://developers.google.com/protocol-buffers/.

[23] *RM0090 Reference manual*. Revision 17. ST Microelectronics. URL: https://www.st.com/resource/en/reference_manual/dm00031020.pdf.

[24] Bruce Schneier. 'COTTONMOUTH-I: NSA Exploit of the Day'. In: *Schneier on Security* (Mar. 2014). URL: https://www.schneier.com/blog/archives/2014/03/cottonmouth-i_n.html.

[25] Bruce Schneier. 'SURLYSPAWN: NSA Exploit of the Day'. In: *Schneier on Security* (Feb. 2014). URL: https://www.schneier.com/blog/archives/2014/02/surlyspawn_nsa.html.

[26] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014. ISBN: 978-1-118-80999-0.

[27] Paul Stoffregen. *Teensy USB Development Board*. URL: https://www.pjrc.com/teensy/.

[28] Dave Tian, Adam Macneil Bates and Kevin Butler. 'Defending against malicious USB firmware with GoodUSB'. In: *Proceedings - 31st Annual Computer Security Applications Conference, ACSAC 2015*. ACM International Conference Proceeding Series. Association for Computing Machinery, Dec. 2015, pp. 261–270. DOI: 10.1145/2818000.2818040.

[29] *TURNIPSCHOOL*. 2015. URL: http://www.nsaplayset.org/turnipschool.

[30] *uinput example code*. Tech. rep. Section 7.4.1: Examples - keyboard events. URL: https://www.kernel.org/doc/html/v4.16/input/uinput.html.

[31] *Universal Serial Bus Specification*. Revision 2.0. USB Implementer's Forum.

[32] *USB Type-C Authentication Specification*. Rev. 1.0. USB Implementers' Forum. Mar. 2016.

[33] *USBNinja*. URL: https://usbninja.com/.

[34] Doug Whiting, Russell Housley and Niels Ferguson. *Counter with CBC-MAC (CCM)*. RFC 3610. RFC Editor, Sept. 2003. URL: https://tools.ietf.org/html/rfc3610.

Appendix A

A.1 UNABRIDGED REPOSITORY OVERVIEW

```
├── README.md - Informational file for the repository
├── common - Code used by both the microcontroller firmware and the host
│   ├── protobuf - Contains the Protocol Buffer files for the protocol
│   │   ├── CMakeLists.txt
│   │   ├── crypto.hpp - Cryptography helper code
│   │   ├── curve25519_donna.[c,h] - Curve25519_donna library
│   │   └── main.cpp
│   ├── protobuf_test_cpp - C++ program: performs basic runthrough of the protocol
│   ├── protobuf_test_python - The Python reference protocol implementation.
│   │   ├── messages*.py - Autogenerated protocol structures
│   │   ├── tsp_decode.py - Helper program: converts binary protocol data to text
│   │   ├── tsp_driver.py - Main Python protocol implementations
│   │   ├── tsp_test.py - Performs a test run of the Python protocol implementation
│   │   └── tsp_utils.py - Utility functions
│   └── protocol_state_machine - The C++ protocol library
│       ├── messages*.[c, h] - Autogenerated protocol structures
│       ├── pb*.[c, h] - nanopb library files
│       ├── tsp_device_state_machine.[cpp, hpp] - Protocol device state machine class
│       ├── tsp_host_state_machine.[cpp, hpp] - Protocol host state machine class
│       ├── tsp_implementation_helpers.[cpp, hpp] - Helper code for above impls
│       └── tsp_state_machine.[cpp, hpp] - Virtual parent state machine class
├── type-safely-firmware - Codebase for the keyboard firmware
│   ├── Makefile - Central Makefile. Also imports platform-specific Makefile
│   ├── libopencm3 - libopencm3 library
│   ├── platforms - Platform-specific implementations live in here
│   │   └── STM32F4
│   │       ├── Makefile.inc - Additional makefile
│   │       ├── external - 3rd party code
│   │       │   ├── cifra - Cifra cryptography library
│   │       │   └── curve25519_donna - Curve25519_donna library
│   │       ├── src - My source code
│   │       │   ├── Platform.[cpp, hpp] - Platform abstraction library header
│   │       │   ├── PlatformAtomic.cpp - Atomicity (interrupt-disabling) functions
│   │       │   ├── PlatformCrypto.cpp - Cryptography functions
│   │       │   ├── PlatformEEPROM.cpp - EEPROM interaction. Based on eeprom_driver
│   │       │   │   by Marco Russi
│   │       │   ├── PlatformPins.cpp - GPIO pin control functions
│   │       │   ├── PlatformPower.cpp - Power management (rebooting etc.)
│   │       │   ├── PlatformSerial.cpp - UART/Serial interaction
│   │       │   ├── PlatformTime.cpp - Time-related functions
│   │       │   ├── PlatformTime_c.[cpp, h] - Exports time functions for C
│   │       │   ├── PlatformUSB.cpp - USB (+keyboard) handling code
│   │       │   ├── PlatformUtils.cpp - Misc. utilities, such as CRC
│   │       │   ├── usb_device.[c, h] - Low-level USB implementation
│   │       │   └── usb_device_config.h - Low-level USB definitions
│   │       └── stm32f415.ld - Linker file for stm32f415. Adopted from libopencm3
│   └── src - Main firmware codebase
```

```
        ├── ASCIIEncodedHIDParser.hpp - Parses hex encoded HID data into bytes
        ├── CircularBuffer.hpp - Custom, interrupt-safe circular buffer
        ├── HIDtoNumberParser.[cpp, hpp] - Parses sequence of keypresses to a number
        ├── KeyboardStateMachine.[cpp, hpp] - Central state machine
        ├── PersistentLTKStorage.[cpp, hpp] - Stores LTKs on EEPROM
        ├── SEGGER_RTT.[c, h] - Logging framework from SEGGER
        ├── USBKeyboardScancodes.h - Defines mapping from USB scancodes to characters
        ├── debug.[cpp, h] - Debugging helper code
        └── main.cpp - Program start point
└── type-safely-daemon - Host daemon codebase
    ├── CMakeLists.txt
    ├── HIDHandler.[cpp, hpp] - Given HID data, sends keystokes to kernel
    ├── HIDLockdownManager.[cpp, hpp] - Configures disabling insecure keyboards
    ├── TSPGUIApplication.[cpp, hpp] - Main application code
    ├── TSPGUIWorker.[cpp, hpp] - GUI-capable device handler thread code
    ├── TSPPairingManager.[cpp, hpp] - Provides thread-safe LTK storage
    ├── TSPPairingWizard.[cpp, hpp] - GUI interface for pairing
    ├── crypto.[cpp, h] - Crypto functions (using Curve25519_donna and OpenSSL)
    ├── curve25519_donna.[c, h] - Curve25519_donna, for Curve25519
    ├── main.cpp - Main code entry point
    ├── modules - CMake modules for cross-platform linking
    ├── pairingpage.[cpp, h] - GUI's pairing page class
    ├── usb.[cpp, h] - USB handling code, using libusb
    └── utils.[cpp, h] - Utility functions
```

A.2 CRUDE LATENCY MEASUREMENT METHOD

To get a feel for the latency introduced by the system, I adapted a method used by Dan Luu [19]: I recorded a video showing a key being pressed and the matching letter appearing onscreen using the 240 fps camera on my iPhone.

I then used an app [2] to count the number of video frames (and therefore time) elapsed before the keypress was registered. I took 20 measurements with the keyboard directly-connected, and another 20 with TypeSafely encrypting packets in-between.

The results are shown in Figure A.1.

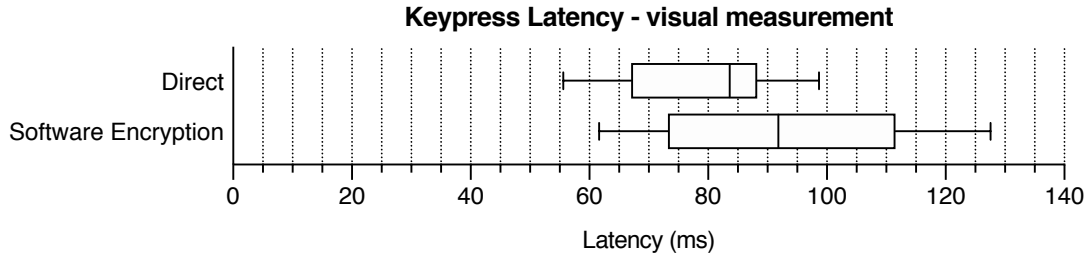


Figure A.1: Box-and-whisker plot showing the latency between a keypress to it being displayed on-screen. Lower is better. Whiskers extend to maximum/minimum values. $n = 20$ for each method.

The overall latency in both scenarios is quite high. This can be attributed to many factors (the standard keyboard's latency, double buffering, etc.), but in this analysis I'm looking at the change in latency, rather than the total. While the high variance reflected the rudimentary experimental setup, the data showed enough of a difference to justify further investigation.

A.3 CODE SAMPLES

Listing A.1: An extract from the Protobuf message specification files

```
syntax = "proto3";

message ECDHPublicKey {
  bytes public_key = 1;
}

message OpenSessionRequest {
  bytes host_authentication_data = 1;
  ECDHPublicKey host_ecdh_public_key = 2;
}

message OpenSessionResponse {
  bytes device_authentication_data = 1;
  ECDHPublicKey device_ecdh_public_key = 2;
}

[...]
message MessageWrapper {
  oneof message {
    [...]
    // Session
    session.OpenSessionRequest
        message_typesafely_protocol_session_OpenSessionRequest = 1015;
    session.OpenSessionResponse
        message_typesafely_protocol_session_OpenSessionResponse = 1016;
    [...]
  }
}
```

Listing A.2: Header extract for the Pins part of the platform abstraction library

```
    namespace Pins {
        typedef enum {
            kPinModeInput,
            kPinModeOutput,
            kPinModeOutputOpenDrain,
        } PinMode;

        void Setup();
```

```
    bool ReadPin(Pin pin);
    void WritePin(Pin pin, bool value);
    void TogglePin(Pin pin);
    void SetPinMode(Pin pin, PinMode mode);

    extern const Pin PinLED;
    extern const Pin PinButton;
}
```

Computer Science Tripos – Part II – Project Proposal

A Secure USB Keyboard

Candidate 2375C

Originator: Dr M. Kuhn

18 October 2018

Project Supervisor: Prof Frank Stajano

Director of Studies: [Redacted per exam regulations]

Project Overseers: Prof G. Winskel & Dr R. Mortier

Introduction

Hardware USB keyloggers and programmable USB keystroke injection tools are widely available, inexpensive, and small enough to be easily concealable. Such devices can even be embedded inside a standard-size USB connector, making their detection nearly impossible. These devices allow the interception of sensitive details and the spoofing of keyboard input, so the threats to computer users can range from theft of sensitive information to the complete compromise of their computer.

This project aims to mitigate these threats by bringing the guarantees of confidentiality, integrity, authentication, and authorisation—provided by protocols such as TLS and Bluetooth—to communication with USB keyboards. This will be achieved by designing and implementing a new protocol, on top of the USB stack, for communication with USB keyboards.

In order to give a reasonable confidence in the security properties of the new protocol, the core of the cryptographic protocol will be based upon an existing, well-studied one, such as Bluetooth's *Simple Secure Pairing*. The implementation will be made from scratch, and will consist of both software for the microcontroller inside the keyboards, and host-side drivers and keyboard management software for Ubuntu Linux.

Starting point

The *USB Human Interface Device* (USB HID) class defines the protocol that standard USB keyboards use. This protocol doesn't include any means of encryption, authentication, or authorisation: the set of keys that are currently pressed is sent over the wire by the keyboard in plaintext.

Much work has been done developing cryptographic protocols that give strong guarantees of confidentiality, integrity, authentication, and authorisation, and techniques and design decisions from existing cryptographic protocols can be used in designing the protocol for

this project. *Bluetooth* is a notable example of a protocol that is designed to provide the desired guarantees, in addition to using a pairing model. However, Bluetooth is designed with consumers and support for a wide range of hardware in mind, so has made some tradeoffs in security in order to improve usability and generality. The project will make different tradeoffs, with an eye to giving stronger guarantees of security and specialisation for keyboards. These decisions can be compared with, and evaluated against, those made in Bluetooth for this specific usecase.

In order to make implementing this project practical, I will make use of a libraries (to be determined) to carry out cryptographic functions (e.g., AES, SHA, ECDH) and abstract away some of the microcontroller hardware for portability. I don't plan on making a keyboard from scratch, and will instead interface with an existing keyboard—likely by implanting a microcontroller inside an existing keyboard.

Resources required

I plan to use my own computer—a macOS machine with 16 GB RAM and a 500 GB SSD—for development. I take regular backups to an external disk and to a cloud provider. The code repositories will be managed using git, and will also be kept in sync with a copy on Github. In the event of failure, I can migrate to a new machine quickly and easily.

I will target Ubuntu Linux as a host machine, and have a suitable virtual machine on my laptop for this purpose. I will make use of an off-the-shelf microcontroller to implement the keyboard itself, and have the necessary tooling to program it. I have spare microcontrollers in the event of failure. I will also need an USB keyboard for testing and to demonstrate my implementation.

Work to be done

The project breaks down into the following sub-projects:

1. Design a protocol that brings the guarantees of confidentiality, integrity, authorisation, and authentication using pairing to keyboards over USB
   (a) Create a suitable attacker model, detailing the what attacks the protocol will aim to defend against (e.g., the project cannot prevent a software keylogger being installed on the host computer)
   (b) Research an existing, well-studied cryptographic protocol that gives guarantees of confidentiality, integrity, authorisation, and authentication using pairing
   (c) Adapt the core of the above cryptographic protocol into a protocol designed to give strong security guarantees to using USB keyboards
2. Implement this new protocol using a microcontroller inside a USB keyboard and a Linux device driver:
   (a) Implement the USB HID keyboard class on the microcontroller to allow it to act as a passthrough from a standard keyboard

(b) Add a profile to the USB interface of the microcontroller code that allows for communicating keystrokes and metadata with a host computer. This will allow for the implementation of the new protocol designed in sub-project 1. Thought will also have to be given to how to ensure that the user is always aware which mode (standard or secure) they're in.

(c) Create a Linux device driver that registers keystrokes from the above new USB class

(d) Modify the host and keyboard implementations to make use of the newly-designed protocol

3. Create a GUI keyboard management/pairing interface to allow administration of the keyboard(s).

4. An evaluation of the success of the project

Success criteria

This project will be a success if I can demonstrate an implementation of a newly-designed protocol that mitigates these makes use of authenticated pairing and encryption for keyboards connected over USB to:

1. prevent passive (e.g., eavesdropper) and active (e.g., man-in-the-middle) keylogging attacks

2. distinguish between keystrokes from a 'trusted' keyboard and those from other sources, allowing the prevention of keystroke spoofing attacks

in an attacker model to be specified as part of the project. The evaluation of part (1) can be aided by analysing the changes from the existing, well-studied protocol, and the use of any test vectors provided as part of the specification of that protocol.

In addition, the project will be evaluated by:

1. An analysis of the realism of the attacker model used

2. Performance: keystroke latency, host CPU/memory usage

3. Friction: how easy is it to set up & pair the keyboard for the first time

Possible extensions

If I achieve my success criteria early, I will attempt to carry out the following extensions:

1. Embedding the microcontroller inside an actual keyboard

2. Proof of knowledge password-based login feature, whereby a user authenticates by typing their password on the keyboard but the keystrokes are not forwarded on to the host computer. Instead, the keyboard transmits a proof that the correct password was entered, preventing software-based keyloggers gaining access to the user's OS login

3. User study on the usability of the complete system
4. Support for other OSes (such as macOS)

Timetable

Planned starting date is 19/10/2018.

- **Michaelmas week 3** (19–24 October)
  Implement the USB HID keyboard class on the microcontroller to allow it to act as a standard keyboard (sub-goal 2a)
- **Michaelmas weeks 4–5** (25 October–7 November)
  Add a profile to the USB interface of the microcontroller code that allows for communicating keystrokes and metadata with a host computer. Create initial Linux device driver that handles this custom profile as a standard keyboard (sub-goals 2b and 2c)
- **Michaelmas weeks 6–7** (8–21 November)
  Complete description of the attacker model, protocol, and pairing procedure created (sub-goal 1)
- **Michaelmas weeks 7–8** (15–28 November)
  Milestone: First proof-of-concept version complete
  Implementation of this new protocol in both the keyboard and the host, as a device driver (sub-goal 2d)
- **Michaelmas vacation** (29 November 2018–16 January 2019)
  Implement a user-facing GUI for management (sub-goal 3), carry out refinement of the code and pairing process
- **Lent week 1** (17–23 January)
  Extend the Linux driver with the ability to ignore keypresses from outside this 'trusted' keyboard (sub-goal 3)
- **Lent weeks 2–3** (24 January–6 February)
  Deadline: Progress Report Deadline - 12 noon, Fri 1 Feb 2019
  Carry out and write up evaluation
- **Lent week 4–Easter week 4** (7 February–17 May)
  Milestone: Lent week 5: Initial draft of dissertation handed off to FMS
  Deadline: Dissertation Deadline - 12 noon, Fri 17 May 2019
  Completion of dissertation. Implementation of project extensions.