Securing the USB interface

Sebastian Götte <secureusb@jaseg.net> @Mori lab, Waseda University

December 12 2018

1 Introduction

1.1 Problem definition

A computer's USB interface is hard to secure. Though overall security is quite good today, the USB interface has not received enough attention. In particular HIDs are a problem, as they are naturally very highly privileged. Off-the-shelf USB HID attack tools exist. In particular from a security point of view extremely bad ideas such as WebUSB[23] are set to increase this already large attack surface even further.

1.2 Contributions

This work includes three key contributions. First, it demonstrates a practical implementation of a complete, backwards-compatible secure USB system using QubesOS and a single new piece of security hardware. Second, it shows a novel interactive user-friendly cryptographic handshaking scheme based on out-of-band communication. Third, it shows and proposes some techniques for the design of general secure protocols that are not limited to USB alone.

2 The state of the art in mitigation

Several ways to secure the USB interface have been proposed that can be broadly categorized as follows.

- USB firewalls are software or hardware that protects the host from requests deemed invalid similar to a network firewall[20, 1, 6, 19, 11].
- USB device authentication uses some sort of user feedback or public key infrastructure to authenticate the device when it connects[3, 4, 22, 5].
- USB bus encryption encrypts the raw USB payloads to ward off eavesdroppers[12, 24].
- For wireless protocols, every conceivable pairing model has been tried. However, not many have been applied to USB[10, 21, 8, 15].
- Compartmentalized systems such as QubesOS separate vulnerable components with large attack surface such as the USB device drivers into VMs to not inhibit exploitation but mitigate its consequences.

	Attacks			**Eavesdropping**		**Backwards compatible**
	HID	**Host exploit**	**Device exploit**	**Bus-level**	**Physical layer**	
Firewalls	○	△	×	△	×	○
Device authentication	○	×	×	△	×	×
Bus encryption	△	×	×	○	○	×
Plain QubesOS setup[1]	△	△	△	△	×	○
Our work	○	○	○	○	○	○

Table 1: Comparison of approaches to USB security

We compare these approaches w.r.t. several attacks in 1. Overall we found that QubesOS is the only advance towards securing this interface that is both *practical* and *effective*. Other approaches have not been successful so far likely due to market inertia and backwards-incompatibility. QubesOS approaches the problem by running a separate VM with the USB host controllers mapped through via IOMMU. This VM runs a linux kernel with a small set of white-listed USB device drivers (HID and mass storage device) and a USB-over-IP backend. A set of Qubes services pass through any HID input arriving inside this VM into dom0, and coordinate exporting USB mass storage devices as Xen block devices. Any other USB devices can be passed-through to other VMs through USB-over-IP-over-QubesRPC, a Xen vChan-based inter-VM communication system. QubesOS is still lacking in that it's compartmentalization becomes essentially useless when it is used with a USB HID keyboard that does not have its own dedicated PCIe USB host controller, as any normal desktop and most recent laptop computers. The issue here is that USB HID is neither authenticated nor encrypted, and the untrusted USB VM sits in the middle of this data stream, which thus allows it trivial privilege escalation via keystroke injection.

2.1 Usage scenarios

Today USB's level security is still adequate for most everyday users. In general, attacks against USB either require special malicious hardware or require re-flashing of existing peripherals with custom malicious firmware. Today's low-level cybercrime targeting everyday users is still focused on much easier tasks such as stealing passwords through phishing, installing cryptolocker malware by means of malicious email attachments and extracting sensitive user data with malicious browser addons. Fortunately, we have not yet entered an age where average computer users need to worry about the type of attack this work defends against. Still, it can be expected that with the general increase of overall computer security, eventually attackers will have to graduate to more advanced means–and since at this time the landscape of effective defenses against USB attacks is very sparse, your author considers it important to explore the avenues to effective defence ealier rather than later in order to be prepared for evolving attacks.

Despite the banality of everyday cybersecurity described above, there already are some people and organizations who face advanced attacks including USB attacks. Due to their exceedingly simple execution, USB HID attacks are a very attractive way to perform targeted

[1]Requires separate USB host controller for HIDs

attacks. For this reason, specialized USB attack hardware is already available commercially at low cost. For users facing targeted attacks like this, SecureHID might already provide practical benefits. The users most at risk of targeted attacks are those either working with highly sensitive data or working with highly privileged access. The former group would include people such as journalists working with their sources and politicians working with confidential information. The latter group would include law enforcement officials, often being endowed with wide-ranging electronic access to databases and other confidential information. Further, system administrators and computer programmers are often provided highly privileged access to critical systems for software deployment using systems such as Ansible or uploading software packages into software repositories such as PyPI.

In all of these scenarios there are many users with very poweful adversaries. In case of a software developer or systems administrator that would be competing companies or foreign intelligence agencies trying to gain access to internal networks to steal confidential information. In case of a journalist that would be whoever they are writing about and here the most interesting articles might come with the most powerful enemies. Finally, a security researcher would by nature of their work, out of academic interest specifically be looking for the most dangerous targets they could find.

Some users might be able to reduce their attack surface to USB attacks by reducing their use of untrusted USB devices, but in many everyday scenarios such as those described above this is not an option. A security researcher needs to connect to untrusted devices in order to analyze them, and using a second, isolated machine for this is very inconvenient. A journalist or politician will frequently have to read USB flash drives with documents for their work, and again simply solving the problem by air-gapping is an effective but impractical mitigation. In all of these cases, SecureHID would be an effective mitigation.

3 Approach

3.1 System overview

The goal of SecureHID is to enable the first reasonably secure system using both HID and arbitrary untrusted devices on the same USB host controller, based on QubesOS. SecureHID consists of a USB HID encryption box to be put between keyboard and computer and a piece of software run inside QubesOS. After initial pairing with the host software, the encryption box will encrypt and sign any USB HID input arriving from the keyboard and forward the encrypted data to the host. The host software running outside the untrusted USB VM will receive the encrypted and signed data from the untrusted USB VM, verify and decrypt it, and inject the received HID input events into Qubes's input event handling system.

A schematic diagram of a system employing SecureHID is shown in figure 1. Two major points that can be seen here are that first, SecureHID requires no specialized hardware on either end and transparently plugs into the existing USB stack. Second, a SecureHID-protected setup has two well-defined security boundaries, one inside the SecureHID device between host and device side, and one in the host operating system between USB driver VM and hypervisor. These security boundaries allow a clean separation of a SecureHID setup into untrusted and trusted domains and greatly simpliefies reasoning about overall system security. Communication across these security boundaries is limited to the simple SecureHID protocol. We describe the design of the SecureHID protocol in section 4.1 and elaborate its security properties in section 4.2. The security of the protocol's core components has been formally verified in the past and the protocol has been kept simple enough to allow exhaustive verification and testing.

3.2 System security properties

This system is sufficient to secure any USB setup, especially unmodified desktop PCs or laptops where a USB host controller is shared between both HIDs and other devices. Attack surface is reduced such that a *full compromise* of the system becomes unlikely, since plain HID is no longer supported. The remaining attack surface consists only of a *compromise of the USB VM*. This attack surface is small enough that other sensitive devices such as USB audio devices can safely be connected. A compromise of the USB driver VM no longer gives full system access, but at best allows listening in on the microphone. Since a compromised USB VM in general does not have network access, such an attack will be mostly harmless in most scenarios. Additionally, the most likely attacking devices would be custom hardware or a compromised smartphone. Custom hardware can easily be outfitted with a microphone, essentially turning it into a bug irrespective of USB functionality, and smartphones already have microphones by definition.

A practical mitigation for potential information leakage through microphones, webcams and other sensitive devices would be to simply unplug them when not used. Microphones could also be connected to a PCIe-based sound card (such as the integrated sound card of most laptops) and webcams could potentially be isolated to a separate USB host controller.

3.3 USB physical-level and bus-level attacks

Since sensitive HIDs are isolated from other USB devices effectively on a separate bus, bus-level attacks such as Neugschwandtner, Beitler, and Kurmus [12] are entirely prevented. The much scarier physical attacks on USB such as Su et al. [18] can be prevented thanks to the clear security boundary inside the SecureHID device. Since there is only four wires needed between the trusted and untrusted sides (Ground, VCC, serial RX and serial TX) and the serial link is running at a comparatively low speed (115.2kBd easily suffice), analog filtering is a viable measure against sidechannels. On the ground and VCC rails extensive filtering using series inductors and large capacitors can be used to decouple both sides. Additionally, both sides' microcontrollers can optionally be fed from separate voltage regulators powered off the USB 5V rail to reduce side-channels. The serial link can be filtered to limit its analog bandwidth to above serial speeds (50kHz) but much below the trusted microcontroller's system clock (72MHz). Finally, on the untrusted microcontroller choosing UART pins that are not multiplexed to its internal ADC elminates the risk of direct measurements by a compromised microcontroller firmware and leaves only indirect measurements of power supplies or coupling into other pins' signals. This means that with a few very inexpensive hardware countermeasures (an additional voltage regulator and a handful of capacitors, inductors and resistors for filtering) any analog side-channels between trusted and untrusted side can be ruled out.

4 Cryptographic design

4.1 Protocol description

The basic protocol consists of two stages: PAIRING and DATA. When the device powers up, it enters PAIRING state. When the host enumerates a new device, it enters PAIRING state. If any

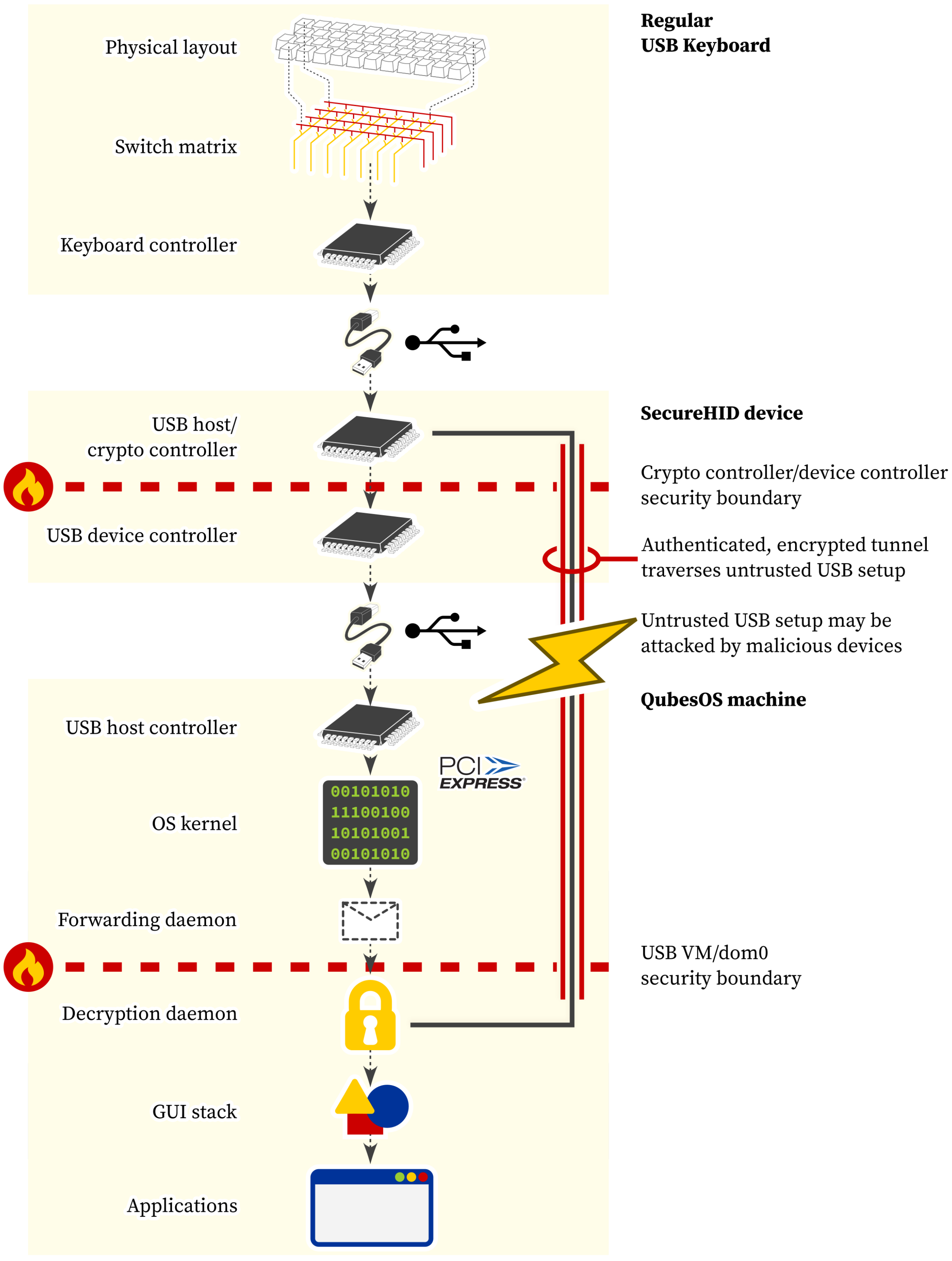

Figure 1: Diagram of a SecureHID-protected system

fatal communication errors occur, both host and device re-enter PAIRING state. To make the implementation robust against host software crashing, devices being unplugged etc. without opening it up to attacks, the host can request the device to re-enter PAIRING state a limited number of times after powerup.

PAIRING state consists of a number of substates as set by Perrin [13]. The device runs noise's XX scheme, i.e. both host and device each contribute both one ephemeral key e and one static key s to the handshake, and the public halves of the static keys are transmitted during handshake encrypted by the emphemeral keys.

The cryptographic primitives instantiated in the prototype are X25519 for the ECDH primitive, BLAKE2s as a hash and ChaCha20-Poly1305 as AEAD for the data phase. ECDH instead of traditional DH was chosen for its small key size and fast computation. Since no variant of RSA is used, key generation is fast. An ad-hoc prototype device-side random number generator has been implemented based on BLAKE2s and the STM32's internal hardware RNG.

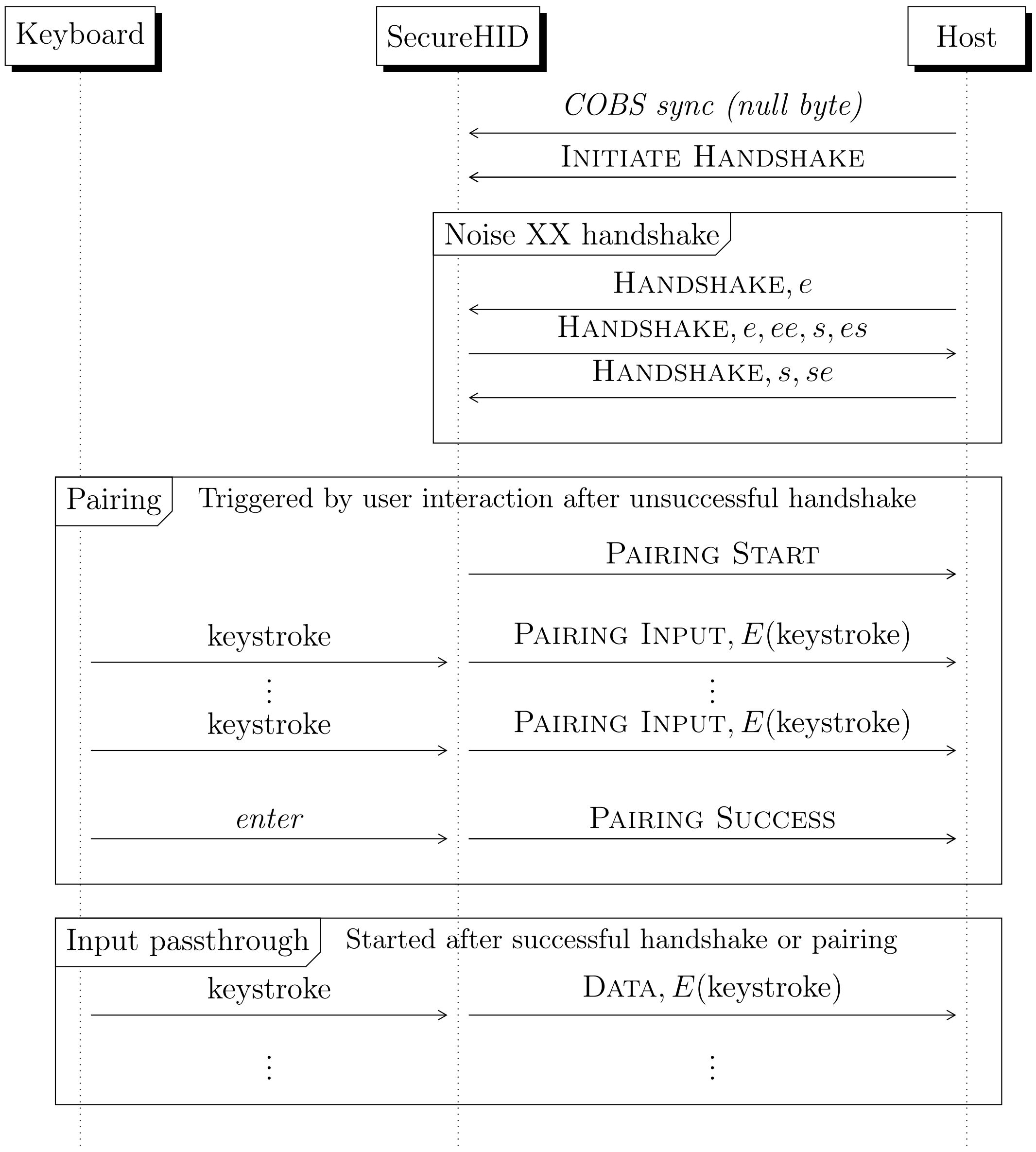

Figure 2: A successful prototype protocol pairing

A successful pairing looks like this:

1. **Handshake.** DEVICE is connected to HOST

2. HOST initiates pairing by sending INITIATE HANDSHAKE to device

3. DEVICE and HOST follow noise state machine for the XX handshake. See figure 3 for a complete flowchart of cryptographic operations during this handshake. The handshake and subsequent Noise protocol communication are specified in Perrin [13] and their security properties are formally verified in Kobeissi and Bhargavan [7]. Section 4.2.1 analyzes the implications of these security properties for this research.

4. After the handshake completes, both DEVICE and HOST have received each other's static public key rs and established a shared secret connection key. At this point, the possibility of an MITM attacker having actively intercepted the handshake remains. At this point DEVICE and HOST will both notice they do not yet know each other's static keys. HOST will respond to this by showing the pairing GUI dialog. DEIVCE will sound an alarm to indicate an untrusted connection to the user.

5. **Channel binding.** Both DEVICE and HOST calculate the *handshake hash* as per noise spec[13]. This hash uniquely identifies this session and depends on both local and remote ephemeral and static keys le, re, ls, rs. Both parties encode a 64-bit part of this hash into a sequence of english words by dictionary lookup. This sequence of words is called the *fingerprint* of the connection.

6. HOST prompts the user to enter the *fingerprint* into a keyboard connected to DEVICE. The user presses the physical pairing button on DEVICE to stop the alarm and start pairing. This step prevents an attacker from being able to cause the device to send unencrypted input without user interaction by starting pairing.

7. As the user enters the *fingerprint*, DEVICE relays any input over the yet-unauthenticated encrypted noise channel to HOST. HOST displays the received user input in plain text in a regular input field in the pairing GUI. This display is only for user convenience and not relevant to the cryptographic handshake. A consequence of this is that a MITM could observe the *fingerprint*[2]. We show in section 4.2 that this does not reduce the protocol's security.

8. When the user has completed entering the fingerprint, the device checks the calculated fingerprint against the entered data. If both match, the host is signalled SUCCESS and DATA phase is entered. If they do not match, the host is signalled FAILURE[3] and PAIRING state is re-entered unless the maximum number of tries since powerup has been exceeded. Failure is indicated to the user by DEVICE through a very annoying beep accompanied by angrily flashing LEDs.

9. **Data phase.** HOST asks the user for confirmation of pairing *in case the device did not sound an alarm* by pressing a button on the GUI. When the user does this, the host enters DATA state and starts input passthrough.

[2]A MITM could also modify the fingerprint information sent from DEVICE to HOST. This would be very obvious to the user, since the fingerprint appearing on the HOST screen would differ from what she types.

[3]Note that this means a MITM could intercept the FAILURE message and forge a SUCCESS message. This means both are just for user convenience *absent* an attacker. If an attacker is present, she will be caught in the next pairing step.

Roughly speaking, this protocol is secure given that the only way to MITM a (EC)DH key exchange is to perform two (EC)DH key exchanges with both parties, then relay messages. Since both parties have different static keys, the resulting two (EC)DH sessions will have different handshake hashes under the noise framework. The channel binding step reliably detects this condition through an out-of-band transmission of the HOST handshake hash to DEVICE. The only specialty here is that this OOB transmission is relayed back from DEVICE to HOST allowing the MITM to intercept it. This is only done for user convenience absent a MITM and the result is discarded by HOST. Since the handshake hash does as a hash does not leak any sensitive information about the keys used during the handshake, it being exposed does not impact protocol security.

4.2 Protocol verifictation

4.2.1 Noise security properties

According to Perrin [13] and proven by Kobeissi and Bhargavan [7] Noise's XX pattern provides strong forward-secrecy, sender and receiver authentication and key compromise impersonation resistance. Strong forward secrecy means an attacker can only decrypt messages by compromising the receivers private key and performing an active impersonation. Strong forward secrecy rules out both physical and protocol-level eavesdropping attacks by malicious USB devices and implies that an attacker can never decrypt past protocol sessions. An implication of the static key checks done on both sides of the connection is that an attacker would need to compromise both host and device in order to remain undetected for e.g. keylogging. Compromising only one party the worst that can be done is impersonating the SecureHID device to perform a classical HID attack. In this case, the attacker cannot read user input and the user would notice this by SecureHID indicating a not connected status and thus the keyboard not working.

To verify that these security properties extend to the overall SecureHID protocol it suffices to show the following three properties.

1. The SecureHID implementation of Noise XX adheres to the Noise specification, i.e. the handshake is performed correctly.

2. Both sides' static keys are verified.

3. All sensitive data is encapsulated in Noise messages after the handshake has ended, and none is sent before.

1 has been validated by manual code review and cross-validation of our implementation against other Noise implementations. 2 has been validated by manual code review. Since all sensitive data in our application is handled on the device in a single place (the USB HID request handling routine), 3 is easily validated by code review. USB HID reports are only transmitted either encrypted after the handshake has been completed or in plain during pairing. Since the host will only inject reports into the input subsystem that have been properly authenticated and encrypted (and not the unauthenticated reports sent during pairing), the protocol is secure in this regard. Since pairing keyboard input is only passed through after the host's pairing request has been acknowledged by the user with SecureUSB's physical pairing button the user would certainly notice an attack trying to exfiltrate data this way. Were pairing input passed through automatically without explicit user acknowledgement, an attacker could start pairing mode just as the user starts typing in a password prompt such as the one of `sudo` or a password

field and might not notice the attack until they have typed out their entire password to the attacker.

4.2.2 Handshake hash non-secrecy

To analyze the impact of disclosing the handshake hash to an adversary we must consider it's definition. The noise protocol specification gives its definition, but does not guarantee that it can be disclosed to an adversary without compromising security. Figure 3 contains a flowchart of the derivation of both initiator-transmit and initiator-receive symmetric encryption keys $k_{1,2}$ and the handshake hash h during the Noise handshake. The definitions of MixHash and MixKey according to the Noise protocol specification are as follows.

$$\text{MixHash}(h, \text{input}) = h' = H(h||\text{input}) \tag{1}$$

$$\text{MixKey}(ck, \text{input}) = (ck', k_{\text{temp}}) = \text{HKDF}(ck, \text{input}, 2) \tag{2}$$

$$\tag{3}$$

Noise's hash-based key derivation function (HKDF) is defined using the HMAC defined in RFC2104[9]. The hash function H employed here depends on the cipher spec used, in this work it is BLAKE2s.

$$\text{HMAC}(K, \text{input}) = H\Big(\left(K \oplus opad\right)||H\left(\left(K \oplus ipad\right)||\text{input}\right)\Big) \tag{4}$$

The HKDF is defined for two and three outputs as follows.

$$\text{HKDF}(ck, \text{input}, n_{\text{out}}) = \begin{cases} (q_0, q_1) & : n_{\text{out}} = 2 \\ (q_0, q_1, q_2) & : n_{\text{out}} = 3 \end{cases} \tag{5}$$

The outputs q_i are derived from chained HMAC invocations. First, a temporary key t' is derived from the chaining key ck and the input data using the $HMAC$, then depending on n_{out} the HMAC is chained twice or thrice to produce $q_{\{0,1,2\}}$.

$$t' = \text{HMAC}(ck, \text{input}) \tag{6}$$

$$\underbrace{\text{HMAC}\Big(t', \underbrace{\text{HMAC}\big(t', \underbrace{\text{HMAC}(t', 1_{16})}_{q_0}||2_{16}\big)}_{q_1}||3_{16}\Big)}_{q_2} \tag{7}$$

Relevant to this protocol implementation's security are the following two properties, both of which can be derived from figure 3:

1. Initiator and responder ephemeral and static keys are all mixed into the handshake hash at least once.
2. Knowledge of the handshake hash does not yield any information on the symmetric AEAD keys k_1 and k_2.

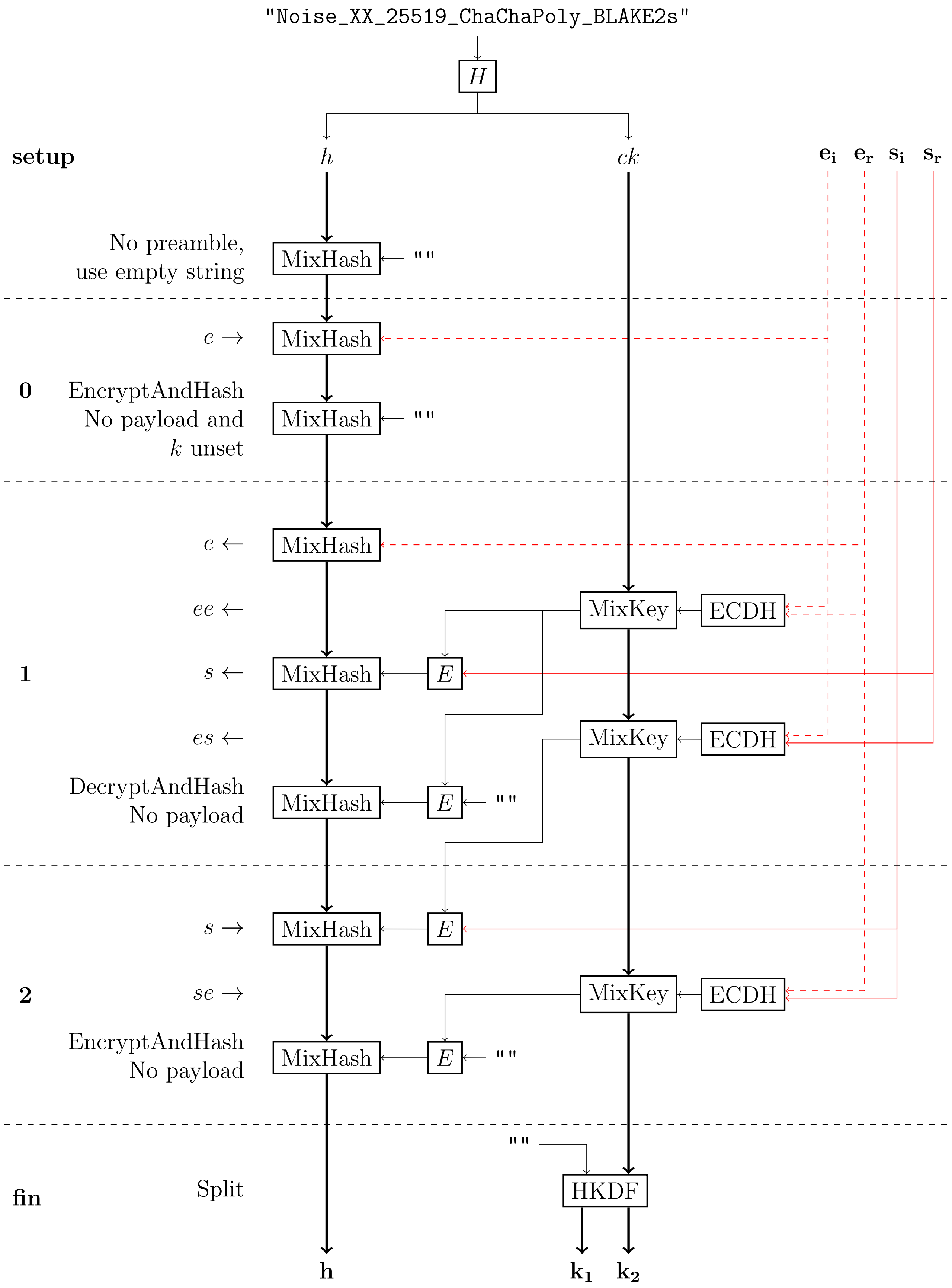

Figure 3: Cryptographic flowchart of Noise XX handshake.

1 is evident since e_i and e_r are mixed in directly and s_i and s_r are mixed in after encryption with temporary encryption keys derived from ck at the $s \rightarrow$ and $s \leftarrow$ steps during the handshake. We can see 2 applies by following the derivation of h backwards. If an attacker learned anything about $k1$ or $k2$ during an attack by (also) observing h that they did not learn before, we could construct an oracle allowing both reversal of H in the final invocation of $MixHash$ and breaking E using this attacker. The attacker would have to reverse H at some point since $h = H(\ldots)$ in the final invocation of MixHash. The attacker would have to recover the key of E in at least one invocation since s_i and s_r are only mixed into h after either being encrypted using E or being used after ECDH to generate a key for E. Since the result of ECDH on e_i and e_r is mixed into h in the $ee \leftarrow$ and following DecryptAndHash steps, h is blinded to an attacker so that they cannot even determine a given k_1 and k_2 match a given h without compromising ECDH security. This means that given the underlying primitives are secure, we do not leak any information on $k1$ or $k2$ by disclosing h.

4.3 Alternative uses for interactive public channel binding

The channel binding method described above can be used in any scenario where a secure channel between two systems must be established where one party has a display of some sort and the other party has an input device of some sort.

Adaption to mice Instead of a keyboard, a mouse can be used for pairing without compromising protocol security. The host would encode the fingerprint bit string into a permutation $\sigma(i) : \{n \in \mathbb{N}, n \leq m\} \rightarrow \{n \in \mathbb{N}, n \leq m\}$ and then display the sequence $\sigma(i)$ in a grid of buttons similar to a minesweeper field with an emualted mouse cursor driven by pairing input on top. The user would then click the buttons on the grid in numeric order. The device would do the same mouse emulation invisible to the user and would be able to observe the permutation this way. The fingerprint can finally be checked by decoding the permutation into a bit string and comparing. The security level for this method in bits is $\eta = \log_2(m!)$ or better than 80bit for $m = 25$ in case of a 5x5 grid. See figure 4 for a mockup of what such a system might look like.

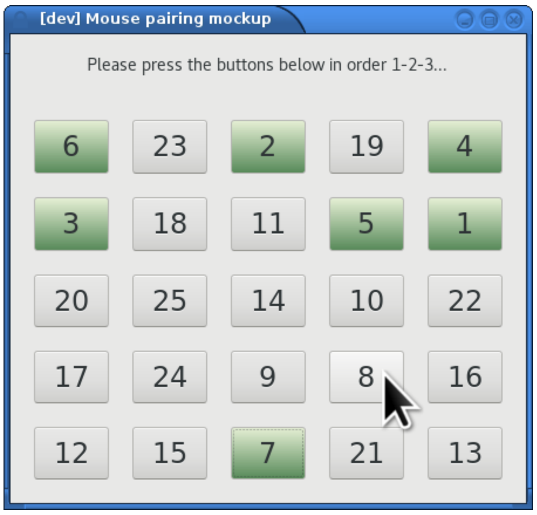
[dev] Mouse pairing mockup
Please press the buttons below in order 1-2-3...
6
23
2
19
4
3
18
11
5
1
20
25
14
10
22
17
24
9
8
16
12
15
7
21
13

Figure 4: A mockup of what mouse-based interactive pairing might look like

Adaption to button input Adaption to button input using few buttons is a little bit harder. The obvious but impractical solution here is to have the user enter a very long numeric string. Entering an 80-bit number on a two-button binary keyboard is not user-friendly. One other option would be to emulate an on-screen keyboard similar to the ones used in arcade and console video games for joystick input. This would be more user-friendly and would likely be a natural interface to many users familiar with such games. One possible attack here is that if the host were to ignore dropped input packets, an attacker might selectively drop packets in order to cause a desynchronization of host and device fingerprint input states. The user would likely chalk up this behavior to sticky or otherwise unreliable keys and while they might find it inconvenient, they might not actually abort the procedure. Thus it is imperative that the host verify there are no dropped packets during pairing. This same observation is also true for keyboard or mouse-based pairing as explained above, but an attack would be much more noticeable there to users as mice and keyboards are generally regarded reliable by most users.

Relation to screen-to-photodiode interfaces There have been many systems using a flashing graphic on a screen to transmit data to a receiver containing a photodiode held against the screen. Such systems have been used to distribute software over broadcast television but have also been used for cryptographic purposes. One widely-deployed example is the "Flicker-tan" system used for wire transfer authorization in Germany where a smartcard reader with five photodiodes is held against a flickering image on the bank website's wire transfer form[16, 17, 2]. Systems such as this one do not benefit from the interactive channel binding process described in this paper since they do not require any direct user interaction. They could however be used as an alternative means for channel binding in any system also supporting interactive pairing as described above. The handshake fingerprint would simply be encoded into the flicker signal and transmitted to the other endpoint in that way. Similarly, QR-codes or other barcodes could be used to a device containing a camera. The primary advantage of photodiode-based systems is that they incur lower implementation complexity and don't require a potentially expensive camera.

5 Hardware implementation

5.1 Hardware overview

To demonstrate the practicality of this approach and to evaluate its usability in an everyday scenario, a hardware prototype has been built. Based on an initial prototype consisting of a microcontroller development board and a bundle of wires a custom PCB fitting an off-the-shelf case has been produced that allows future usability testing in practical settings.

The hardware implementation consists of two ARM microcontrollers, one for the untrusted host side and one for the trusted device side. Both are linked using a simple UART interface. Both microcontrollers have been chosen by their USB functionality. For the integrated USB host controller, we had to chose a rather powerful microcontroller on the trusted device side even though a much less powerful one would have sufficed even though we are doing serious cryptography on this microcontroller. AES encryption is done on every data packet and must compelete in time for the overall system to meet its latency requirement, but is fast enough by a large margin. Similarly, the hash and ECDH operations during the cryptographic handshake are fast enough by a large margin. Additionally, those operations are only invoked infrequently any time the device is disconnected or the host suspends.

5.2 Hardware security measures

5.3 Usability considerations

6 Evaluation

7 Future work

8 Conclusion

References

[1] Sebastian Angel et al. “Defending against Malicious Peripherals with Cinch”. In: *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016), pp. 397–414.

[2] Lars-Dominik Braun. “chipTAN Flickercodes”. In: (2012). URL: `https://web.archive.org/web/20181213014441/https://6xq.net/flickercodes/`.

[3] Bob Dunstan, Abdul Ismail, and Stephanie Wallick, eds. *Universal Serial Bus Type-C Authentication Specification.* 2017.

[4] Federico Griscioli, Maurizio Pizzonia, and Marco Sacchetti. “USBCheckIn: Preventing BadUSB Attacks by Forcing Human-Device Interaction”. In: (2017).

[5] Debiao He et al. “Enhanced Three-factor Security Protocol for Consumer USB Mass Storage Devices”. In: *IEEE Transactions on Consumer Electronics* 60.1 (Feb. 2014), pp. 30–37.

[6] Myung Kang and Hossein Saiedian. “USBWall: A novel security mechanism to protect against maliciously reprogrammed USB devices”. In: *Information Security Journal "A Global Perspective"* 26.4 (2017), pp. 166–185.

[7] Nadim Kobeissi and Karthikeyan Bhargavan. “Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols”. In: (Dec. 2018). URL: `https://eprint.iacr.org/2018/766.pdf`.

[8] Alfred Kobsa et al. “Serial Hook-ups: A Comparative Usability Study of Secure Device Pairing Methods”. In: *Symposium on Usable Privacy and Security (SOUPS)* (July 2009).

[9] H. Krawczyk, M. Bellare, and R. Canetti. *RFC2104 - HMAC: Keyed-Hashing for Message Authentication.* Feb. 1997.

[10] Arun Kumar et al. “Caveat Emptor: A Comparative Study of Secure Device Pairing Methods”. In: (2009).

[11] Edwin Lupito Loe et al. “SandUSB: An Installation-Free Sandbox For USB Peripherals”. In: (2016).

[12] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. “A Transparent Defense Against USB Eavesdropping Attacks”. In: *EUROSEC’16* (Apr. 2016).

[13] Trevor Perrin. *The Noise Protocol Framework.* Tech. rep. Rev. 34. July 2018.

[14] Srivaths Ravi et al. “Security in Embedded Systems: Design Challenges”. In: *ACM Transactions on Embedded Computing Systems* 3.3 (Aug. 2004), pp. 461–491.

[15] Nitesh Saxena et al. “Secure Device Pairing based on a Visual Channel”. In: *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006).

[16] *TAN-Generatoren mit optischer Schnittstelle (Flickercode)*. 2009. URL: `http://web.archive.org/web/20130309011417/http://www.hbci-zka.de/dokumente/spezifikation_deutsch/Belegungsrichtlinien%20TAN-Generator%20ve1.3%20final%20version.pdf`.

[17] Andreas Schiermeier. *Vom Überweisungsauftrag zur TAN*. 2018. URL: `https://web.archive.org/web/20181213014203/https://wiki.ccc-ffm.de/projekte:tangenerator:start`.

[18] Yang Su et al. “USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs”. In: *Proceedings of the 26th USENIX Security Symposium* (Aug. 2017), pp. 1145–1161.

[19] Dave (Jing) Tian, Adam Bates, and Kevin Butler. “Defending Against Malicious USB Firmware with GoodUSB”. In: *ACSAC* (Dec. 2015).

[20] Dave Tian et al. *Making USB Great Again with USBFILTER*. Austin, Texas, Aug. 2016.

[21] Ersin Uzun, Kristiina Karvonen, and N. Asokan. *Usability Analysis of Secure Pairing Methods*. Tech. rep. Helsinki, Finland: Nokia Research Center, 2007.

[22] Zhaohui Wang, Ryan Johnson, and Angelos Stavrou. “Attestation & Authentication for USB Communications”. In: (2012).

[23] *WebUSB API*. 2018. URL: `https://wicg.github.io/webusb/`.

[24] David Weinstein, Xeno Kovah, and Scott Dyer. “SeRPEnT: Secure Remote Peripheral Encryption Tunnel”. In: (2012).

A Project state

A working prototype has been completed.

A.1 Completed

- Rough protocol design
- Protocol implementation based on Perrin [13] using noise-c (microcontroller) and noise-protocol (python/host)
- SRAM-based key storage with SRAM wear levelling
- host/device signature checking
- host/device key generation
- proper circuit design because I was bored last weekend (see appendix **??**)

A.2 Open issues

- Both noise-c and noiseprotocol have poor code and API quality. Since most noise functionality is not needed, just implement the protocol in bare C/python based on cryptographic primitives and scrap higher-level protocol implementations (though they've been useful so far during prototyping).
- Implement HID mouse host support
- Test USB hub support
- Replace the serial link with a custom USB link using an STM32F103 instead of the CH340G USB/serial converter
- Properly integrate prototype host client with qubes infrastructure
- Implement photodiode/monitor-based pairing side-channel

B Possible directions

- Possibly contrast to carmera/other backchannel systems
- Elaborate possible DisplayPort/HDMI-based display encryption → Bunnie's NeTV2 w/ HDMI/eDP converter
- Elaborate possible encrypted remote input (SSH) setups
  - This might turn out to be really interesting
  - For this to be usable the host needs to tell the device at least which keyslot to use which could turn out to be complex to implement securely
  - Considering complexity, this might turn into its own research project
- Showcase secure hardware interface design, contrast with wireguard protocol design
  - Formally derive handshake security properties
  - Formally derive host/device protocol security properties using noise spec
  - Formally verify and thouroughly unit-test the host/device protocol implementation on all layers
  - IMHO this is the most interesting part of this project from an engineering point of view
- Create custom hardware prototype
- Benchmark cryptography routines (will likely turn out to be "wayyy fast" for HID, fast enough for full-speed USB. High-speed cannot be done with the current architecture as we can't get data out the chip at high-speed data rates. Ravi et al. [14] raise the issue of running crypto on embedded systems, but in this case it turns out with somewhat modern hardware and cryptography there is no problem at all.