社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

SecureHID: USB インタフェースのセキュリティ

ゲッテ　ヤン[†]　矢内　直人[††]　森　達哉[†,†††]

† 早稲田大学大学院 基幹理工学研究科　〒169–8555 東京都新宿区大久保 3–4–1
†† 大阪大学大学院 情報科学研究科　〒565–0871 大阪府吹田市山田丘 1-5
E-mail: †code@jaseg.net, ††yanai@ist.osaka-u.ac.jp, †††mori@nsl.cs.waseda.ac.jp

あらまし　USB インターフェースは世の中で広く使われているため，そのセキュリティ対策を確立することは重要な意義がある．これまで多数の USB セキュリティ技術が提案・開発されてきたが，互換性の欠如や不十分なセキュリティ保証など，未解決の課題がある．本論文は Noise プロトコルフレームワーク [15] を用いてバス通信を暗号化することにより，USB セキュリティを実現する汎用的な方法を提案する．提案手法は既存の USB デバイスとの互換性を確保し，また誤った利用方法によって攻撃が成功してしまうリスクを避けるための直感的なユーザインタフェースを提供する．仮想 OS である QubeOS と統合された形で動作するハードウェアプロトタイプ実装を通じて，提案手法の有効性を実証する．

キーワード　USB, ハードウェアセキュリティ, バス暗号化, 仮想化, USB HID 攻撃, 防御

SecureHID: Securing the USB Interface

Jan GOETTE[†], Naoto YANAI[††], and Tatsuya MORI[†,†††]

† Deapartment of Computer Science, Waseda University　3–4–1 Okubo, Shinjuku-ku, Tokyo, 169–8555 Japan
†† Information Security Engineering, Multimedia Engineering, Osaka University　1–5 Yamada, Suita-shi, 565–0871 Japan
E-mail: †code@jaseg.net, ††yanai@ist.osaka-u.ac.jp, †††mori@nsl.cs.waseda.ac.jp

Abstract　The USB interface poses an increasing security concern. Various security techniques have been proposed with disadvantages ranging from incompatibility with existing devices to insufficient security guarantees. In this paper, we propose a novel approach to generic USB security using bus encryption based on the Noise protocol framework [15]. Our approach is compatible with any existing USB device and provides an intuitive user interface reducing the risk of accidential compromise. We demonstrate the viability of our approach with a fully-working hardware prototype with integration into QubesOS.

Key words　USB, hardware security, bus encryption, virtualization, USB HID attacks, defense

1. Introduction

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[24] are set to increase this already large attack surface even further.

Several ways to secure the USB interface have been proposed. USB firewalls are software or hardware that protects the host from requests deemed invalid similar to a network firewall [21, 1, 8, 20, 13]. USB device authentication uses some sort of user feedback or public key infrastructure to authenticate the device when it connects [3, 5, 23, 6]. USB bus encryption encrypts the raw USB payloads to ward off eavesdroppers[14, 25]. Eskandarian et al.

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

表 1 Comparison of approaches to USB security

[4] propose a more high-level system that protects certain types of sensitive application data such as payment information based on trusted execution (SGX). For wireless protocols, every conceivable pairing model has been tried. However, not many have been applied to USB [12, 22, 10, 16]. As we shall discuss in short, although these countermeasures mitigate a certain kinds of attacks, they cannot mitigate entire attack vectors against USB interface.

With this background in mind, this work provides the following three key contributions. First, we present a practical implementation of a complete, backwards-compatible secure USB system using QubesOS and a single new piece of security hardware. Second, we provide a novel interactive user-friendly cryptographic handshaking scheme based on out-of-band communication. Third, we provide some techniques for the design of general secure protocols that are not limited to USB alone. The key idea of our approach is to leverage the compartmentalization provided by the QubeOS[7]. QubesOS is a hypervirtualized operating system using Xen to keep *domains* of an user's digital life in separate Linux or Windows-based virtual machines while providing full GUI integration. Prior to our work, QubesOS cannot easily be secured against malicious USB devices.

2. Background

2.1 Comparison between the existing approaches and our work

We compare several approaches to USB interface security in Table 1. Overall we found existing systems to lack either *practicality* or *effectiveness*. The most prominent issue seems to be backwards-incompatibility. Presently, QubesOS approaches USB security by mapping the USB host controller into a dedicated Linux VM. HID input is passed from this VM into dom0. Mass storage devices are forwarded as Xen block devices. Other USB devices are passed through using USB-over-IP over Xen vChan. USB HID devices pose a security challenge here since a compromised USB VM could emulate any HID input if USB HID is enabled.

2.2 Threat Model

The security level of today's USB may be adequate for most everyday users. USB attacks require malicious hardware or firmware. The bulk of cybercrime is phishing, banking troyans and ransomware. With the general advance of computer security, eventually attacks will have to advance too. Our target is to prepare for such evolved attacks.

Everyday cybersecurity being fairly mundane there are people and organizations facing advanced attacks. Exceedingly simple USB HID attacks are an attractive way to perform such targeted attacks and specialized attack hardware is commercially available at low cost. For users working with highly sensitive data such as journalists or politicians as well as users with highly privileged access such as law enforcement officials or system administrators our approach might already yield practical benefits. We are concerned with very powerful adversaries. A software developer or systems administrator might have to defend against competing companies or foreign intelligence agencies. A journalist might be targeted by whoever they are writing about–the most interesting articles might come with the most powerful enemies.

Some can reduce their attack surface by not using untrusted USB devices, but in many scenarios this is not an option. A security researcher needs to connect to untrusted devices for analysis and a journalist or politician has to use USB flash drives with documents for their work. Air-gaps solve this problem but are impractical. Our work provides an effective mitigation.

3. System Specification

3.1 System overview

The goal of our work is to enable the first reasonably secure system using both HID and arbitrary untrusted devices on the same USB host controller(注2), based on QubesOS (Figure 1). After initial pairing, in our setup USB HID requests are encrypted by a security device between keyboard and computer and authenticated and decrypted by a piece of software inside the QubesOS dom0. Here, HID requests are passed through the untrusted parts of the USB stack (hardware and the driver VM) inside AEAD.

Our setup transparently plugs into the existing USB stack and works with any USB device and host. This setup

(注1) Requires separate USB host controller for HIDs

(注2) Many laptops only have one USB host controller, and using a separate trusted controller for HID devices might not be practical.

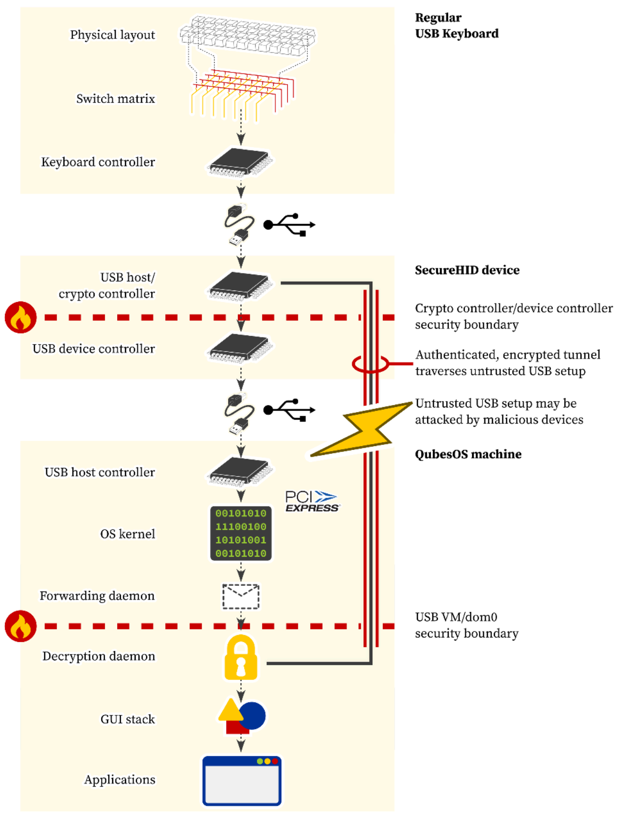


図 1 Diagram of a system secured using our approach.

has two well-defined security boundaries: One inside the security device and one inside the hypervisor surrounding the USB driver VM. These security boundaries allow clean separation of trusted and untrusted components, simplifying reasoning about overall system security. Communication across security boundaries is limited to the easy-to-audit protocol described in section 4. 1 and analyzed for security in section 4. 2.

3. 2 System security properties

Our system is sufficient to secure any USB setup such as a desktop PCs or a laptop sharing a USB host controller between privileged HID and other unprivileged devices. *Full compromise* of the system becomes unlikely due to limited attack surface. A possible *compromise of the USB driver VM* would not pose a large risk anymore. Considering a scenario where a sensitive USB audio device is connected to the USB driver VM, an attacker cannot escalate their privileges into dom0 from there anymore and since the USB VM does not have network access this compromise would be harmless in most scenarios.

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 [14] are entirely prevented. The much scarier physical attacks on USB such as Su et al. [19] can be prevented thanks to the clear security boundary inside the security 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 sidechannels. 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

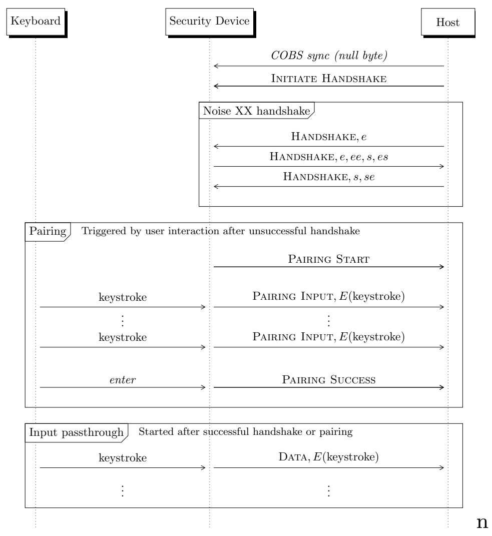
The basic protocol consists of two stages: PAIRING and DATA. When the device powers up, it enters PAIRING state. When the host enumerates a new device, it enters PAIRING state. If any fatal communication errors occur, both host and device re-enter PAIRING state. To make the implementation robust against 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 [15]. The device runs the *Noise 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 static public keys are transmitted during handshake encrypted by the emphemeral keys.

A successful pairing looks like this:

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

(2) HOST initiates pairing by sending INITIATE HAND-



n

図 2　A successful prototype protocol pairing

SHAKE to device

（3） DEVICE and HOST follow noise state machine for the *Noise 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 [15] and their security properties are formally verified in Kobeissi and Bhargavan [9]. 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[15]. This hash uniquely identifies this session and depends on both local and remote ephemeral and static keys le, re, ls, rs. Boteh 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*[注3]. 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[注4] and PAIRING state is reentered 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.

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 protocol security properties

According to Perrin [15] and proven by Kobeissi and

[注3] 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.

[注4] 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.

Bhargavan [9] the *Noise 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 security device to perform a classical HID attack. In this case, the attacker cannot read user input and the user would notice this by the security device indicating a not connected status and thus the keyboard not working.

4. 2. 2 Implementation correctness

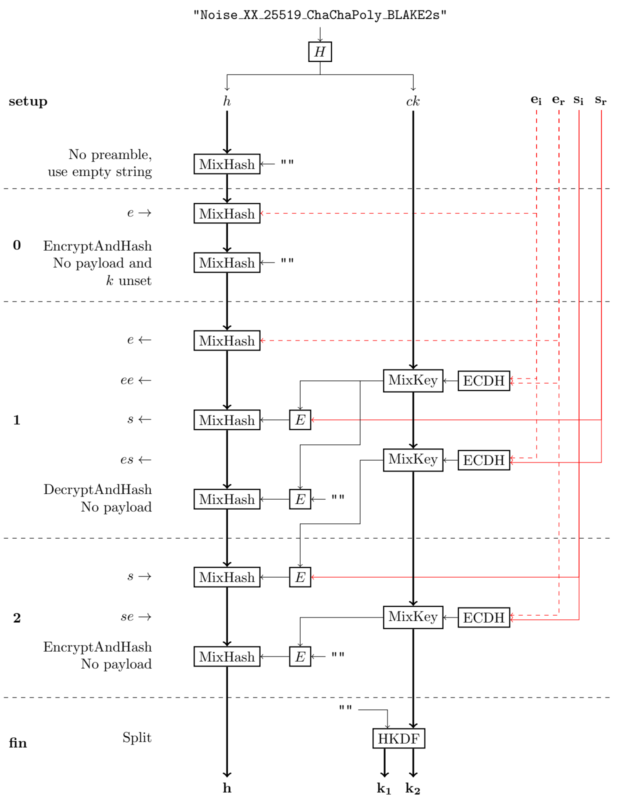
To verify that these security properties extend to our implementation it suffices to show the following three properties.

（1） Our implementation of *Noise XX* adheres to the Noise specification, i.e. the handshake is performed correctly.

（2） Both sides verify each other's static key.

（3） All sensitive data is encapsulated in Noise messages after the handshake has ended, and none is sent before. I.e. no sensitive data is transmitted outside the Noise protocol.

1 has been validated by manual code review and cross-validation of our noise-c-based implementation against noiseprotocol, a noise implementation in python. 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 the physical pairing button the user would certainly notice an attack trying to exfiltrate data this way. Were pairing input passed through



"Noise_XX_25519_ChaChaPoly_BLAKE2s"
H
setup
h
ck
e_i e_r s_i s_r
No preamble, use empty string
MixHash
""
e →
0
EncryptAndHash No payload and k unset
e ←
ee ←
MixKey
ECDH
1
s ←
E
es ←
DecryptAndHash No payload
2
s →
se →
EncryptAndHash No payload
fin
Split
HKDF
h
k1 k2

図 3 Cryptographic flowchart of Noise XX handshake.

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. 3 Handshake hash non-secrecy

To analyze the impact of disclosing the handshake hash to an adversary we must consider its definition. The noise protocol specification does not guarantee that the handshake hash can be disclosed to an adversary without compromising security. Figure 3 is 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. Following are the definitions of MixHash and MixKey according to the Noise protocol specification.

$$\begin{aligned} \mathrm{MixHash}(h, \mathrm{input}) &= h' = H(h \| \mathrm{input}) \\ \mathrm{MixKey}(ck, \mathrm{input}) &= (ck', k_{\mathrm{temp}}) = \mathrm{HKDF}(ck, \mathrm{input}, 2) \end{aligned}$$

Noise's hash-based key derivation function (HKDF) is defined using the HMAC defined in RFC2104[11]. The hash function H employed here depends on the cipher spec used. In this work we use BLAKE2s.

$$\mathrm{HMAC}(K, \mathrm{input}) =$$

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

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

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

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 $q_{\{0,1,2\}}$.

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

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

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.

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, an attacker cannot even determine a given k_1 and k_2 match a given h without reversing ECDH. This means that if the underlying primitives are secure, we do not leak any information on $k1$ or $k2$ by disclosing h.

5. Implementation

5.1 Hardware prototype

To demonstrate the practicality of our approach we built a hardware prototype of the security device (Figure 4). Our prototype consists of two ARM microcontrollers with USB peripherals, one for the untrusted host side and one for the trusted device side. Both are linked using a simple UART interface. Despite the tough cryptography on the trusted microcontroller the primary constraint was not its computational power but its USB host peripheral.

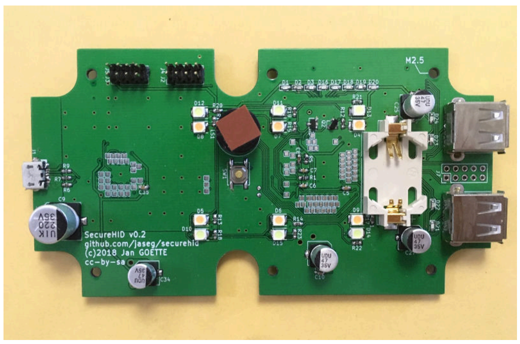


図 4 Hardware prototype PCB front view

5.2 Hardware security measures

To provide some level of defence against physical-level USB attacks as outlined in Su et al. [19] the security device hardware prototype uses separate voltage regulators for its trusted and untrusted sides. In addition, the PCB layout is prepared to allow easy insertion of additional conducted EMI filtering on the two communication and two power lines between the trusted and untrusted sides if practical measurements show a problem there.

5.3 Cryptographic parametrization

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.

5.4 Usability considerations

a） Implementation robustness

A common problem is that overall system security heavily depends on implementation details such as certificate checking, and user interface details such as the precise structure of security warning messages. The complexity of these components in practice often leads to insecure systems, such as a system using TLS checking a certificate's internal validity but omitting checks on the certificate's chain of trust. A nice property of the key estabilishment system outlined in this paper is that it is both very simple, reducing surface for errors and it tightly couples the critical channel binding step during key establishment to the overall system's user interface. In a system using either keyboard or mouse-based interactive channel binding, an implementation that does not perform the channel binding step correctly would simply not work. If the host does not display the correct fingerprint the user cannot enter it and the device will not complete the binding step. If the device does not relay fingerprint data correctly during pairing the host application would clearly indicate to the user things are amiss with the input not matching the fingerprint. Since the channel fingerprint is computed in a cryptographically well-defined way based on entropy contributed by both partners during pairing a implementer would not even be able to accidentially degrade fingerprint security.

The critical point from an UI perspective in this pairing scheme is that the host application must display correct instructions to the user for them to complete pairing. It should make sure the user actually checks whether the device raised an alarm before confirming pairing after fingerprint input. If it didn't the user would eventually notice their keyboard and mouse not functioning, but an attacker might have gained unauthorized access in the meantime. The device needs a clearly understandable method of indicating pairing failure to the user. In practice a loud buzzer and a few bright LEDs are a good solution.

6. Discussion

In this section, we discuss future research topics that need to be explored on top of the SecureHID framework.

6. 1 Variations on the pairing technique

a) Screen-to-photodiode interfaces

Numerous systems use a flashing graphic on a screen to transmit data to a photodiode held against the screen, e.g. to distribute software over broadcast televisiong. One widely-deployed system is the "Flickertan" 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[17, 18, 2]. This approach could be used as a non-interactive alternative to the pairing protocol described above.

b) Adaption to mice

Instead of a keyboard, a mouse could also be used for pairing. In a basic scheme, the host would encode the fingerprint into a permutation $\sigma(i) : \{n \in \mathbb{N}, n \leq m\} \rightarrow \{n \in \mathbb{N}, n \leq m\}$ for an integer security parameter $m > 0$ and then display the sequence $\sigma(i)$ in a grid of buttons with an emulated mouse cursor. The user would then click the buttons on the grid in numeric order. Invisible to the user, the device would emulate the cursor and observe the permutation this way. The fingerprint is then checked against this permutation. If m is the number of grid buttons used this method provides a security level of $\eta = \log_2(m!)$ bits. A 5-by-5 grid yields better than 80bit security.

c) Gamification

A second mouse-based approach would be to adapt Minesweeper to compare fingerprints in our setting. The host would encode the fingerprint into the minesweeper field, then let the user play the game using their mouse. The security device would forward all mouse input while internally emulating the mouse pointer. A the end of the game the security device knows which locations the user marked as mines and can decode the fingerprint from this..

Minesweeper is well-known and can be parametrized to be easily solved by most people. The number of ways to place n mines on a x by y field is $\binom{xy}{n}$, the number of unordered subsets of n elements. The default difficulties in by Windows XP minesweeper provide ≈ 40 to ≈ 348 bit of entropy. Two rounds at beginner difficulty or one round at intermediate difficulty already suffice for an 80 bit security level.

d) Adaption to button input

Adaption to button input using few buttons is complex. Entering an 80-bit number on a two-button binary keyboard is not user-friendly. A potentially user-friendly option would be to emulate an on-screen keyboard similar to the ones used in arcade and console video games for joystick input. One possible attack here is that if an attacker can selectively drop packets they can cause a desynchronization of host and device fingerprint input states. To the user this might seem like unreliable keys and they might not actually abort the procedure. This observation is also true for keyboard or mouse-based pairing as explained above, but packet loss would be much more noticeable there.

6. 2 Alternative uses for interactive public channel binding

a) Adaption for SSH identity distribution

Our interactive channel binding method using a passphrase could be used for key establishment in an SSH setup. SSH includes a simple public-key mutual authentication system, but does not include key management functionality. In most cases the user will have to provide their own identity management layer on top of the primitives provided by SSH.

The interactive channel binding method described in this paper could be used to interactively transfer an SSH key's public to another host by establishing a secure channel, then transferring the key through it. This would allow two users to transfer a key by simply reading out aloud the channel binding fingerprint. This reduces the key exchange problem to the problem of two users being sure whether they're actually talking to each other instead of an impostor.

A second scenario using the security device to improve SSH security would be to terminate the SSH connection inside the security device. This would reduce the scope of host compromise but would pose much new firmware attack surface. Additionally this approach would only work for keyboard input and would break things like scp or scripted SSH as used in configuration management systems such as ansible.

b) Continuous authentication

systems analyze a user's behavior such as mouse movements and keystroke timings to detect malicious activity. Using the security device as a trust anchor to authenticate user input against a monitoring server could be used to build a remote continuous authentication system, placing no trust in the (potentially compromised) user's machine.

7. Conclusion

In this paper, we have demonstrated a fully working prototype of a system secured against all known USB attacks based on QubesOS and a simple hardware device. We have elaborated on protocol design as a key component of overall system security. We have shown that an intuitive interactive pairing process built on top of the Noise protocol framework satisfies all our security requirements.

We outlined several directions for future work. A user experience study could explore variations on the pairing scheme. Our SSH key exchange concept is interesting for systems administration. We have done an informal security arguments in this paper. This argument should be formalized before practical deployment of a security system like this.

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] Saba Eskandarian et al. "Fidelius: Protecting User Secrets from Compromised Browsers". In: *CoRR* abs/1809.04774 (2018). URL: `http://arxiv.org/abs/1809.04774`.

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

[6] 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.

[7] Rafal Wojtczuk Joanna Rutkowska. *Qubes OS Architecture.* 2010. URL: `https://www.qubes-os.org/attachment/wiki/QubesArchitecture/arch-spec-0.3.pdf`.

[8] 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.

[9] 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`.

[10] 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).

[11] H. Krawczyk, M. Bellare, and R. Canetti. *RFC2104 - HMAC: Keyed-Hashing for Message Authentication.* Feb. 1997. URL: `https://tools.ietf.org/html/rfc2104`.

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

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

[14] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. "A Transparent Defense Against USB Eavesdropping Attacks". In: *EUROSEC'16* (Apr. 2016).

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

[16] 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).

[17] *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`.

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

[19] 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.

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

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

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

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

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

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