Smart-Guard: Defending User Input from Malware

Denzel, Michael; Bruni, Alessandro ; Ryan, Mark

Smart-Guard: Defending User Input from Malware

Michael Denzel*, Alessandro Bruni†, and Mark D. Ryan*

*University of Birmingham, School of Computer Science,
B15 2TT Birmingham, United Kingdom
m.denzel AT cs.bham.ac.uk, m.d.ryan AT bham.ac.uk

†IT-University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
brun AT itu.dk

***Abstract*—Trusted input techniques can profoundly enhance a variety of scenarios like online banking, electronic voting, Virtual Private Networks, and even commands to a server or Industrial Control System. To protect the system from malware of the sender's computer, input needs to be reliably authenticated. Previous research in this field is based on fixed assumptions about trustworthy components and is, thus, too rigid for this use case.**

We present Smart-Guard, a method to protect user input into a system even if the attacker controls – to us unknown – parts of the underlying system. Our approach ensures integrity of user input even when up to two of three devices are compromised; confidentiality holds for one malicious device. In this way, Smart-Guard has flexible trust assumptions, and does not require any particular part of the system to be trusted. To prove our claims, we formally verified our protocol using the state-of-the-art protocol verifier ProVerif. Additionally, we define a new class of techniques, *malware tolerance*, which operate securely even when the system is infected with malware.

1. Introduction

The term *trusted input* is defined as the problem of securing user input from device end-point (e.g. a keyboard) to program end-point. *Trusted output* specifies the stream in the other direction (program to device). Both terms are also subsumed by the expression *trusted path* or *trusted I/O* and belong to the wider domain of *trusted execution*. [1], [2]

Trusted input techniques are already utilised to harden the infrastructure in fields such as online banking and electronic voting but are also applicable to Virtual Private Networks (VPNs) or commands to a server (e.g. via SSH) or to an Industrial Control System (ICS). Authentication of the exact origin of the input is essential in all of these scenarios as the following thoughts demonstrate:

- Recent online banking trojans (ZitMo [3]) spread from PCs to smart-phones to compromise two-factor authentication. Afterwards they impersonate the banking customer.
- Malicious or bogus commands to ICSs can damage these systems (see e.g. Stuxnet [4], [5]) and potentially lead to catastrophes especially since ICSs are used in the energy, transport, and health sector.
- Keyloggers are able to intercept login credentials and hijack connections. [6]
- After compromising an administrator account in a company network, an adversary can pretend to be the administrator and gain widespread access to assets and resources. [7]
- Forged, i.e. wrongly authenticated, votes in electronic voting compromise the entire voting system. [8]

To improve on these scenarios we need to exactly identify the input source, that means we have to be able to distinguish between authorised users and a potentially compromised PC.

Current approaches [2], [9], [10], [11], [12], [13] (detailed explanation in section 5) for trusted I/O usually omit hardware attacks which means firmware attacks, keyloggers and similar are still effective. Moreover, they make strong and inflexible assumptions about which parts of the system are trustworthy. For critical resources like a power plant or electronic votes, we need stronger security guarantees and cannot purely rely on the trustworthiness of a single component.

We introduce Smart-Guard, which secures input even if malware controls some parts of the system in real time. Smart-Guard distributes trust over several components of the system, and remains secure even if one or more of them are compromised. No single component is required to be invulnerable to attack; an attack on one component can be resisted if other components are trustworthy. This means that we do not rely on a single trusted computing base, but allow flexibility about the trust assumptions.

Contributions:

- We propose Smart-Guard, a trusted input technique that is reliable even when an attacker controls some of its components in real time (see section 2). Our system guarantees integrity and authentication when at least one (out of three) components is not controlled by the attacker. It can also ensure confidentiality under stricter conditions.
- We provide a formal proof of our security claims for Smart-Guard, using a state-of-the-art protocol verification tool called ProVerif (section 4.2).
- We define *malware tolerance*, a new class of techniques (section 4.3), which provides security properties even if the underlying system is compromised.

2. Overview of Smart-Guard

Smart-Guard is a trusted input system consisting of a computer, a smart-card, and an encryption-capable keyboard[1], i.e. a keyboard with built-in encryption module which encrypts typed data. Keyboard and smart-card (smart-card reader) are both connected to the computer via USB or similar.

One could argue that an encryption-capable keyboard could simply encrypt or sign keystrokes by itself, but this would only shift the trust from the operating system driver to the keyboard. We want to achieve stronger assurances and, thus, need to distribute trust upon multiple devices to tolerate localised attacks. An adversary would have to compromise more than one device to be successful (sections 4.1 and 4.3 will give details). In particular, Smart-Guard can resist (confidentiality and integrity) an infected computer, provided that keyboard and smart-card are not compromised.

Basic procedure of Smart-Guard: Consider a scenario where an authorised user wants to send a message or a command to a recipient from his or her commodity computer. The recipient could be the user's bank, an election server, or the flow control system of an oil pipeline.

The user would type the characters of the message into the encryption-capable keyboard which sends them signed and encrypted to the smart-card to prevent the computer from tampering with the data. The smart-card verifies the keystrokes by displaying them via any form of confidentiality preserving output, like e.g. ARM TrustZone with a TrustZone-aware screen. When typing is finished, the user confirms the displayed input by entering a short string.

Smart-card and keyboard each produce one partial signature of the message. Those partial signatures can only be combined into a valid signature if the two devices agree on the same input. The combined cryptographic signature is verified by the recipient.

Objective: Our primary goal is to protect integrity and authentication of the user-given keystrokes. The recipient of the keystrokes should be able to verify the origin of the input. The system must at least guarantee these security properties if one of the three participating devices (user PC, smart-card, keyboard) is malicious. We also provided confidentiality under stricter conditions but the main focus lies on authenticating the input.

Our assumptions are:

1) The attacker can control some unknown parts of the user's system. These can be entire devices such as the computer or the keyboard. However, the attacker cannot control the whole Smart-Guard system. Thus, our goal is to tolerate at least 1 (best case: 2) malicious devices of the 3 devices.
   This seems realistic since we grant the attacker access to our infrastructure. Compromising PC, keyboard, and smart-card involves physical access to the building and theft of the user's smart-card which would not scale and would, moreover, pose a great risk.
2) For keyboard and smart-card, we assume that they have no additional communication channels apart from the ones to the computer (e.g. USB). This could be verified by inspecting the devices. Note that this does not forbid a separate hardware keylogger.
3) The user is a trusted entity. Authentication (password, biometrics) additional to the smart-card has to take place beforehand to prevent untrusted persons from signing in with stolen or lost cards.
4) The recipient is trusted and has an asymmetric key pair, with a known public key. This research does not focus on trusted execution of the recipient system.

1. e.g. https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRFready-Desktop-2-Reference-Design

3. Smart-Guard Protocol

Our protocol consists of three phases:

1) An input phase which distributes user input to the devices. This step runs for every input character.
2) A transition phase marking the end of input.
3) A signature and encryption phase which only takes place once per message.

We will now introduce the protocol phases one by one. They are also displayed in Fig. 1 and Fig. 2.

Setup: (1) The encryption-capable keyboard and the smart-card share a symmetric key k_{ks} which pairs the two devices. (2) Both also receive a partial key (respective k_1 and k_2) of a *mRSA* [14] algorithm.

Mediated RSA (mRSA) is an asymmetric cryptosystem which splits the private key into two shares $k_{priv} = k_1 + k_2$. This way, signatures can be created out of two partial signatures ps_1 and ps_2 (see Eq. 1). Note that the plaintext m has to be appropriately padded (details in [14]).

$$signature = ps_1 * ps_2 = m^{k_1} * m^{k_2} = m^{k_1+k_2} = m^{k_{priv}} \ . \tag{1}$$

Phase 1: Every character the user types into the keyboard is encrypted with a stream cipher and sent to the PC. The PC forwards them to the smart-card which decrypts them. Smart-card and keyboard buffer the characters for later signing. At this stage, only the keyboard verified the input. To allow the user to confirm the typed keystrokes, the smart-card displays them via any form of confidentiality preserving output to hide them from the computer. Confidential output can be provided, for example, by an ARM TrustZone enabled device. Note that even if the secrecy of the *output* is compromised, the integrity of the *input* is unaffected (assuming no further colluding attacks).

Phase 2: After the user indicated that he wants to end the input (e.g. by a mouse click), the smart-card creates a keyed hash of the input and displays it via confidential output. The user then types this verification string into the keyboard. The keyboard can only forward these characters via the PC to the smart-card as they appear random. This way the smart-card can independently verify that the user agrees with the keystrokes. A keyed hash has two advantages: (1) it cannot be forged by keyboard or PC and (2) it is clear to which command it corresponds.

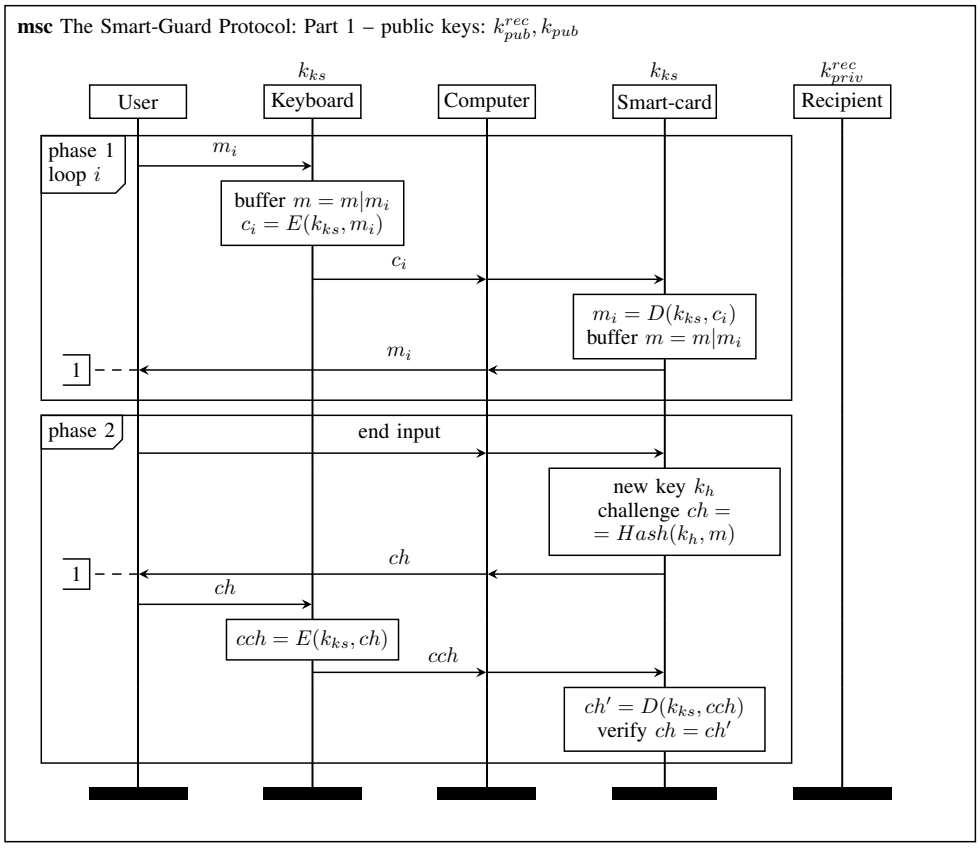


Figure 1: *The Smart-Guard Protocol - Part 1:* The user types in keystrokes m_i which are distributed to the encryption-capable keyboard and the smart-card. An end-input event (e.g. a mouse click) indicates that the user wants to proceed. The user verifies the keystrokes, which are displayed via secure output (label 1), by typing a challenge into the keyboard. Afterwards phase 3 (see Fig. 2) is run.

$S(k, m)$ denotes a signature of data m with key k, $E(k, m)$ is an encryption, $D(k, c)$ is a decryption, and $m_i|...|m_0$ represents a concatenation. The secret keys are displayed over each device at the top of the diagram.

Phase 3: The keyboard and the smart-card generate a fresh key via a Diffie-Hellman key exchange on an encrypted channel. The keyboard encrypts the user input and creates a partial mRSA signature. Both are delivered to the smart-card for verification. When the smart-card verified the encryption, it will add the second part of the mRSA signature to complete the structure which can then be sent to the recipient.

The three phases are the logical steps to create a shared, signed ciphertext. Phase 1 distributes the input to the participating devices which sign and encrypt the input in phase 3. The second phase seems unnecessary at first but prevents the keyboard from appending characters to the message.

Important details: It is worth mentioning some important details in the protocol which are essential to achieve the security guarantees. These details are labelled 1 – 4 in Fig. 1 and 2 and will now be explained.

1) For confidentiality, the input should be displayed with a technique for confidential output (integrity protection is not needed).
2) The Diffie-Hellman key exchange is executed via the encrypted and authenticated channel of k_{ks} (e.g. AES-GCM) to prevent Man-in-the-Middle attacks. The key exchange generates a fresh key k'.
3) The smart-card recalculates the RSA encryption of k' to prevent the keyboard from adding additional keys. A malicious keyboard could e.g. send $\overline{r}_1 = RSA(k_{pub}^{attack}, k')$. This cannot be distinguished from the original $r_1 = RSA(k_{pub}^{rec}, k')$ without recomputing it and forces us to do so. For this, it is necessary to send the random parameters c_{IV} (encrypted) to the smart-card.
4) The smart-card creates the signature and immediately verifies it afterwards. This reveals malicious behaviour of the keyboard.

4. Security Analysis

4.1. Models

Adversary model: We assume the attacker has already full control over some parts of the user's system and we are unaware which parts these are.

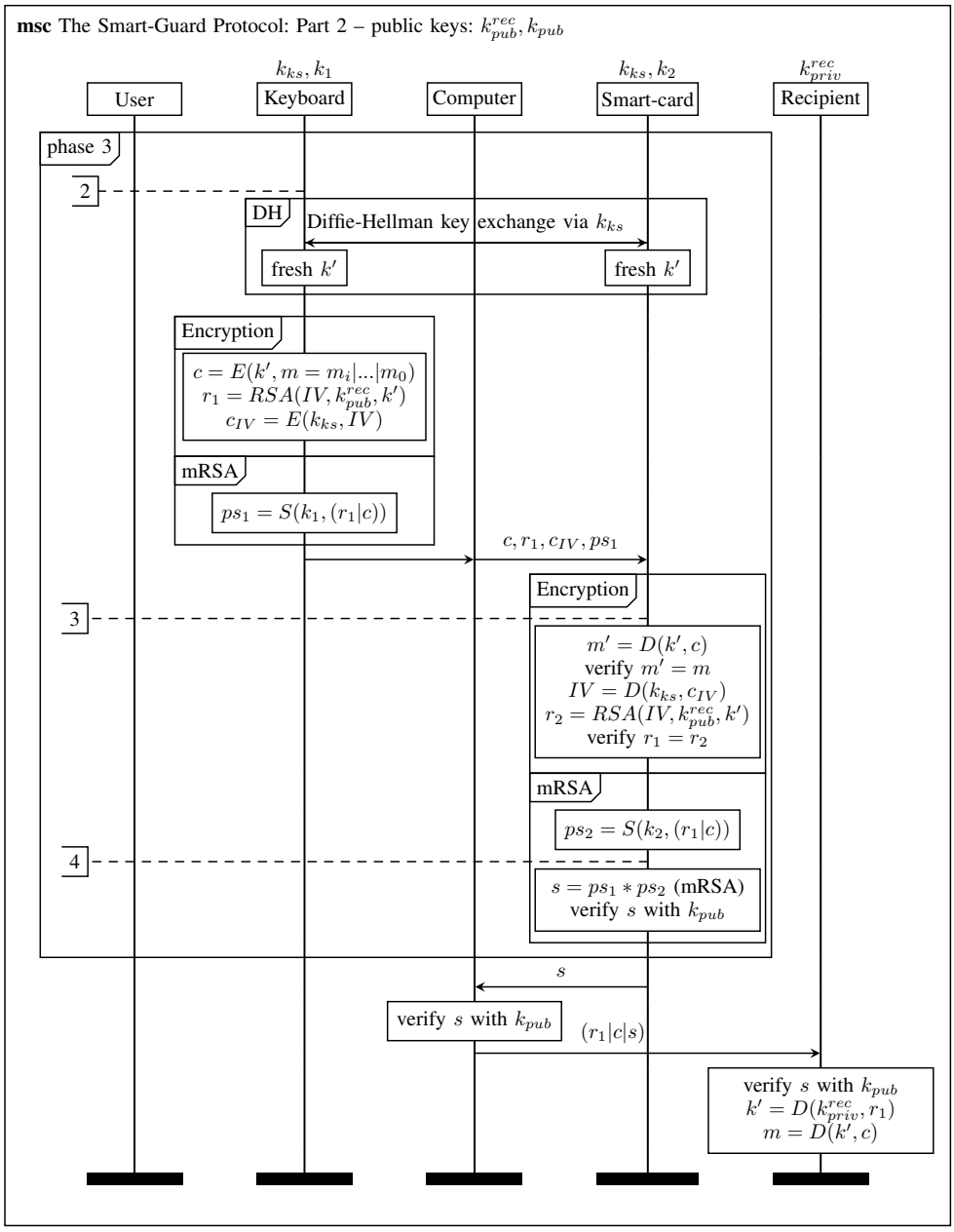


Figure 2: *The Smart-Guard Protocol - Part 2:* After phase 2 (see Fig. 1) the keyboard and the smart-card are supplied with a verified copy of the user input. Encryption-capable keyboard and smart-card now generate a fresh, shared key k'. The keyboard encrypts the input with k' and adds a partial mRSA [14] signature, which is completed by the smart-card when verifying the encryption.

TCB Model: In contrast to the classical single Trusted Computing Base (TCB) model, our model consists of several TCBs with flexible trust assumptions. If any 1 (of N) TCBs is secure, the system is secure. In other words, TCBs are *OR*'ed together. If TCB_1 is secure *OR* TCB_2 is secure then the system is secure. An adversary has to compromise all TCBs to be successful which is significantly harder than attacking a single TCB.

The TCBs for the Smart-Guard protocol are defined in Table 1. To e.g. compromise integrity, an attacker has to gain control over two TCBs. Notice that the PC is not part of any of them which means that its security is irrelevant for integrity.

Integrity:	Confidentiality:
$TCB_1 =$ { smart-card }	$TCB_1 =$ { PC, smart-card }
$TCB_2 =$ { keyboard }	$TCB_2 =$ { smart-card, keyboard }
	$TCB_3 =$ { PC, keyboard }

TABLE 1: Trusted Computing Bases

The TCB model for integrity can be interpreted as shown in Eq. 2.

$$\begin{aligned} \text{keyboard secure} &\Rightarrow \text{integrity} . \\ \text{smart-card secure} &\Rightarrow \text{integrity} . \end{aligned} \tag{2}$$

The confidentiality side of Table 1 is equal to Eq. 3. Assuming a secure technique for trusted output, the attacker needs to control three TCBs to compromise confidentiality. This corresponds to two of three devices as every device is part of two TCBs.

$$\begin{aligned} \text{PC secure} \wedge \text{smart-card secure} &\Rightarrow \text{confidentiality} . \\ \text{smart-card secure} \wedge \text{keyboard secure} &\Rightarrow \text{confidentiality} . \\ \text{PC secure} \wedge \text{keyboard secure} &\Rightarrow \text{confidentiality} . \end{aligned} \tag{3}$$

4.2. Proofs

We proved our protocol using ProVerif[2], a state-of-the-art protocol verifier. In ProVerif, protocols are represented with *messages* which are sent over public or private *channels* between *processes*. During a protocol execution user-defined *events* can happen. We can formulate predicates, so-called *queries*, which ProVerif will try to verify. To do so, ProVerif has a built-in *attacker* who is able to participate on public channels. The following section will introduce our proofs.

In contrast to classical ProVerif proofs, we work with two attackers: one is controlling some of the three devices (PC, keyboard, smart-card), while the other one passively waits for messages. We call them the offline and online attacker respectively. This characteristic simulates the fact that smart-card or keyboard cannot communicate to the network without the PC (see assumptions in section 2). As a result, the compromised devices (i.e. the offline attacker) have to trick the PC to communicate to the second attacker – assuming the PC is honest. A malicious PC is modelled by making all communication channels of it public in ProVerif.

Due to the proofs for the protocol becoming fairly long, we split them accordingly to Fig. 1 and 2 into two parts. We will now explain a few elements of the proofs. The source code and all required files are available online[3].

4.2.1. Proofs for Phase 1 and 2. The results of phase 2 define the pre-conditions of phase 3. We need to test three properties:

- Does the keyboard have the correct input? (message integrity keyboard)
- Does the smart-card have the right input? (message integrity smart-card)
- Does the protocol protect the input from an online attacker? (confidentiality)

To verify confidentiality, we introduce another property called "strong confidentiality" which tests if the offline attacker gets the input (makes four properties to test).

Either the keyboard or the smart-card have to know the correct message in order to proceed to phase 2. Integrity is, therefore, the union of message integrity at the keyboard and message integrity at the smart-card. We formulated the four properties as ProVerif queries in Fig. 3.

```
(* integrity *)
query m:char;
  event (sc_pass(m)) ==>
          event (user_pass(m)).
query m:char;
  event (kb_pass(m)) ==>
          event (user_begin(m)).

(* confidentiality *)
query mess(ch_att, new m).
(* strong confidentiality *)
query attacker(new m).
```

Figure 3: *ProVerif Queries Phase 1 and 2:* The code excerpt defines four queries:
Query 1: If the smart-card accepts the message m (event sc_pass), the user must have accepted it on the screen output (event user_pass).
Query 2: If the keyboard accepts the message m (event kb_pass), the user must have created that message (event user_begin).
Query 3: The offline attacker does not get the message m.
Query 4: The online attacker does not receive message m.

4.2.2. Proofs for Phase 3. In phase 3 we assume the results of phase 2. This should be that the keyboard or the smart-card have a user-verified copy of the message. Message m might be compromised by the offline attacker but was not sent to the online attacker. The protocol will create a joint ciphertext and signature. We verified that the signature at the recipient side is indeed correct and that the online attacker cannot retrieve the plaintext of the message m (see Fig. 4).

```
(* integrity *)
query m:char;
  event (rec_end(m)) ==>
          event (kb_begin(m)) ||
          event (sc_begin(m)).

(* confidentiality *)
query mess(ch_att, new m).
```

Figure 4: *ProVerif Queries Phase 3:* The queries are:
Query 1: If recipient accepts message m (event rec_end), it came from the keyboard (event kb_begin) or from the smart-card (event sc_begin).
Query 2: The online attacker does not receive the message m. (or: the offline attacker cannot communicate m via channel *ch_att*)

4.2.3. Combined Results. The results of our ProVerif queries are shown in Table 2. Important for this part is that either smart-card or keyboard have a valid message (integrity column) and that the attacker on the internet did not receive the message (confidentiality column).

We summarised the results in Fig. 5a and 5b. Each circle in the Venn diagram represents that a device is trusted while the complement stands for the device being compromised.

2. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
3. https://github.com/mdenzel/smartguard

Trust			Confidentiality	Integrity	End
PC	Smart-card	Keyboard			reached
honest	honest	honest	✓	✓	✓
honest	honest	untrusted	✓	✓	✓
honest	untrusted	honest	✓	✓	✓
untrusted	honest	honest	✓	✓	✓
untrusted	untrusted	honest		✓	✓
untrusted	honest	untrusted		✓	✓
honest	untrusted	untrusted	(✓)		✓
untrusted	untrusted	untrusted			✓

TABLE 2: *ProVerif Results:* Brackets indicate that a property does not hold in phase 3 of the protocol.

The grey area marks cases where our protocol satisfies the particular property labelled below the figure. Smart-Guard guarantees integrity for either a trustworthy keyboard or a trustworthy smart-card (Fig. 5a) and confidentiality for one malicious device out of three devices (see also Fig. 5b).

We tested if Smart-Guard defends against hardware keyloggers with ProVerif. A hardware keylogger corresponds to the channels between PC and the other two devices (usually USB channels) being public. ProVerif verified these queries and, thus, Smart-Guard defends against hardware keyloggers in some cases (see dotted area in Fig. 5a and 5b).

4.2.4. Performance Estimation. To estimate performance we use the crypto benchmark of Dai [15] and assume a smart-card CPU with 5 MHz.

Phase 1: According to Dai AES/CTR (128-bit key) needs 12.6 clocks per byte which would take $12.6/5\text{MHz} = 2.52\mu s$ per byte on a 5 MHz CPU. As we need to encrypt and decrypt the keystrokes, we roughly need $2 * 2.52\mu s = 3.04\mu s$ per byte. This is equivalent to an average typing speed of $60s/3.04\mu s = 19,736,842$ characters per minute.

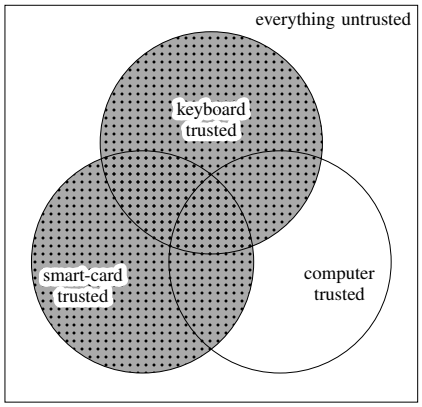
Phase 2: The main bottleneck for the short verification string is the delay imposed by the user. Thus, the cryptographic performance is negligible.

Phase 3: To generate the resulting ciphertext, our protocol executes (in order): a Diffie-Hellman key exchange, an AES encryption, a RSA encryption (for the key), an AES encryption of the initialisation vector, a RSA signature, an AES decryption, an AES decryption of the initialisation vector, a RSA encryption, a RSA signature, and a RSA verification. When assuming a message of 1024 bytes, we need $0.82/5s + 2.52*1024\mu s + 0.14/5s + 2.52*128\mu s + 2.71/5s + 2.52*1024\mu s + 2.52*128\mu s + 0.14/5s + 2.71/5s + 0.13/5s = 1.34s$. Notice that this is only executed once per message and could run in the background.

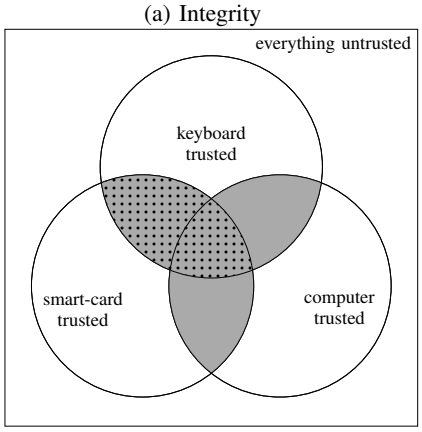
These numbers are a rough estimate and have to be considered carefully. However, they indicate that performance is not a major concern.

4.3. Malware Tolerance

As demonstrated by Smart-Guard, using malicious devices is possible under certain conditions. We grant the adversary access to the system and even let him or her choose the compromised devices to some extent. Integrity



(a) Integrity



(b) Confidentiality

Figure 5: *Trust Model for Smart-Guard:* The "keyboard trusted" circle represents the cases in which the keyboard is trusted, and similarly for the other two circles. Thus, the very centre of the diagram means that all devices are trusted.

Grey area: protocol satisfies integrity (Fig. 5a) or confidentiality (Fig. 5b).
Dotted area: protocol defends against hardware keyloggers.

is guaranteed if either keyboard or smart-card are honest while confidentiality holds provided two devices are honest. When the communication on all connections to and from the PC are encrypted (which is similar to smart-card and keyboard being honest), then even hardware keyloggers cannot compromise the message. That means Smart-Guard can defend against a compromised PC which colludes with a hardware keylogger.

The basic idea is that trust could be distributed upon several, independent devices which interact in such a way that the individual device cannot meaningfully tamper with the data or resources. Thus, the method tolerates devices which are compromised by malware. We named this class of techniques *malware tolerance* (Tech. Rep. [16], [17]).

An adversary would have to compromise multiple devices to successfully attack the system which is significantly more difficult than manipulating a single device. Intuitively, the upper bound of malicious parts to tolerate is at maximum $N-1$; one part has to be honest in order to detect or prevent

attacks of the others and report to the user.

To formally examine these techniques, we introduced the multiple TCB model. Each TCB groups devices which are able to correctly produce the desired result even when all other devices are compromised (section 4.1). A technique is malware-tolerant, if several such TCBs exist. When one TCB is compromised, another honest TCB would automatically detect or prevent attacks of the first one.

5. Related Work and Comparison

We identified other approaches which could be called malware-tolerant. Examples for protocols are online banking with hardware tokens [19] and electronic voting [20]. There is also a special compiler [21] which restricts attackers to public method calls of the programming language. Other work includes hardware to strengthen hypervisors while protecting guest virtual machines [22] and a two-way sandbox called Minibox [23] which protects the host Operating System (OS) as well as the sandboxed guest-application. Vasudevan et al. [24] defined five properties such secure systems should fulfil: isolated execution, secure storage, remote attestation, secure provisioning, and a trusted path. Rijswijk and Deij [25] suggested to add local attestation to the user as sixth property. Basin et al. [26] analysed general topologies of setups for human-computer communication.

Table 3 gives an overview of the existent techniques for trusted path and compares them based on the information the papers provided. We will give a brief overview now:

1) *BitE* – McCune et al. [11]: *Bump in the Ether (BitE)* combines a PC, an encryption-capable keyboard, and a mobile phone to achieve a trusted path. User input is sent from the keyboard to the trusted phone which forwards the keystrokes (encrypted) to the OS. There, they are distributed to the correct applications. The trusted phone also serves as trusted output.
2) *Bumpy* – McCune et al. [12]: McCune et al. used a computer with Flicker [27] in combination with a special keyboard to deliver passwords securely to a recipient (e.g. webserver). A phone serves again as output to indicate the receiving application and the webserver. The technique consists of two stages: collecting the keystrokes and encrypting or hashing them in order to sent them to the recipient.
3) *UTP* – Filyanov et al. [10]: The authors worked together with McCune to realise trusted input with Flicker. Their approach, Uni-directional Trusted Path (UTP), is aimed at CAPTCHAs and confirmations for banking transactions. UTP switches temporarily away from the OS to Flicker which securely displays a confirmation dialog and signs the user's input. The result is sent to the bank where it is verified.
4) *DriverGuard* – Cheng et al. [9]: In contrast to other techniques, DriverGuard aims to achieve a trusted path in software and shields I/O drivers with a hypervisor. So-called Privileged Code Blocks (PCBs) have access rights to I/O resources while accesses of the guest OS are denied.
5) *KeyScrambler* – QFX Software [13], [28]: The commercial tool KeyScrambler intercepts keystrokes at the keyboard driver, encrypts them during processing of the OS, and decrypts them in the actual application. The difference to DriverGuard is that KeyScrambler only handles input but manages without a hypervisor. It partly defends against user space keyloggers.
6) Zhou et al. [2]: The authors of TrustVisor [29] created a user-verifiable trusted path. The approach uses TrustVisor to shield drivers at a lower level than DriverGuard protecting I/O ports, device memory access, device configuration space, and interrupts. Additionally, the authors enabled the user to verify the system with a custom hand-held device with a red/green indicator LED and a Trusted Platform Module (TPM). The hand-held device requests attestation from the TPM about the honesty of the platform. The technique defends against software attacks while maintaining usability during run-time.
7) *TrustZone* [18]: TrustZone is a Trusted Execution Environment (TEE) of ARM CPUs. The system is split into two zones, normal and secure world, with different privileges managed by the so-called monitor. Combined with TrustZone-aware devices it can provide Trusted I/O if the entire TrustZone architecture is trusted.
8) *Smart-Guard*: Our technique defends against software, hardware, and even attacks of the own devices. The TCB model differs from previous research in the fact that there are flexible trust assumptions: trust is distributed over multiple TCBs of several devices. If one of the devices is compromised, the others will prevent attacks automatically (section 4.1). The limiting factor is that Smart-Guard relies on other techniques for trusted output.

6. Conclusions

We presented Smart-Guard, the first malware-tolerant protocol to protect user input from malware and hardware attacks. It is especially useful to authenticate user input. Smart-Guard consists of three devices – a PC, a keyboard, and a smart-card – and guarantees security properties even in the context of attacks of one of the underlying devices. We designed the protocol to be secure under several sets of trust assumptions, providing flexibility and avoiding a single point of failure making it malware tolerant. To provide evidence of our claims, we formally verified the protocol with ProVerif and analysed its security properties. Our work proves that several devices can interact in a way that prevents any individual device from compromising the resources; we called this *malware tolerance*.

For the future, we plan to examine ARM TrustZone for its capabilities to improve our technique.

Acknowledgments

This work was partially supported by the EPSRC project "Analysing security and privacy properties".

Technique	Trusted input	Trusted output	Confidentiality	Integrity	Software attacks	Hardware attacks	Trusted Computing Base
BitE [11]	✓	$\sim^1$	✓	✓	$\sim^2$		keyboard, phone, PC, OS
Bumpy [12]	✓	$\sim^1$	✓	✓	✓		keyboard, Flicker, TPM
UTP [10]	✓			✓	✓		keyboard, Flicker, TPM, hypervisor
DriverGuard [9]	✓	✓	✓	✓	$\sim^3$		I/O devices, PC, hypervisor
KeyScrambler [13]	✓		✓	✓	$\sim^2$		keyboard, PC, OS
Zhou et al. [2]	✓	✓	✓	✓	✓		hypervisor, hand-held device, TPM
TrustZone [18]	✓	✓	✓	✓	$\sim^4$		secure world, monitor, bootloader, TrustZone, I/O devices
Smart-Guard	✓	$\sim^5$	✓	✓	✓	✓	multiple/flexible (see section 4.1)

[1] limited by phone [2] only user-space malware blocked [3] attacks of drivers possible [4] secure world/monitor attacks possible [5] relies on other trusted output techniques

TABLE 3: *Comparison of trusted path techniques:* Similar techniques are grouped together (dashed line). Ticks indicate the referred property is secure while a tilde means that it has drawbacks.

References

[1] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, vol. 13, 2013.

[2] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Symposium on Security and Privacy (SP)*. IEEE, 2012.

[3] Kaspersky, "Teamwork: How the zitmo trojan bypasses online banking security," Online, October 2011, accessed: 2015-02-16. [Online]. Available: http://www.kaspersky.com/about/news/virus/2011/Teamwork_How_the_ZitMo_Trojan_Bypasses_Online_Banking_Security

[4] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE*, vol. 9, pp. 49–51, Nov 2011.

[5] B. Zhu, A. Joseph, and S. Sastry, "A taxonomy of cyber attacks on scada systems," in *4th international conference on cyber, physical and social computing*. IEEE, 2011.

[6] N. Virvilis, D. Gritzalis, and T. Apostolopoulos, "Trusted computing vs. advanced persistent threats: Can a defender win this game?" in *10th International Conference on Ubiquitous Intelligence and Computing and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*. IEEE, 2013.

[7] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, p. 80, 2011.

[8] A. J. Feldman, J. A. Halderman, and E. W. Felten, "Security analysis of the diebold accuvote-ts voting machine," in *USENIX Electronic Voting Technology Workshop*, 2006.

[9] Y. Cheng, X. Ding, and R. H. Deng, "Driverguard: A fine-grained protection on I/O flows," in *Computer Security–ESORICS*. 2011.

[10] A. Filyanov, J. M. McCune, A.-R. Sadeghiz, and M. Winandy, "Uni-directional trusted path: Transaction confirmation on just one device," in *IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011.

[11] J. M. McCune, A. Perrig, and M. K. Reiter, "Bump in the ether: A framework for securing sensitive user input," in *USENIX Annual Technical Conference*, 2006.

[12] ——, "Safe passage for passwords and other sensitive data," in *16th Annual Network and Distributed System Security Symposium (NDSS)*, February 2009.

[13] QFX software, "How keyscrambler works," Online, 2015, accessed: 2015-05-25. [Online]. Available: https://www.qfxsoftware.com/ks-windows/how-it-works.htm

[14] D. Boneh, X. Ding, G. Tsudik, and C.-M. Wong, "A method for fast revocation of public key certificates and security capabilities," in *USENIX Security Symposium*, 2001.

[15] W. Dai, "Crypto++ 5.6.0 benchmarks," online, Mar 2009, accessed: 2016-03-11. [Online]. Available: https://www.cryptopp.com/benchmarks.html

[16] M. Denzel and M. Ryan, "Malware tolerance," University of Birmingham, Tech. Rep. 1, March 2014

[17] ——, "Malware tolerance," University of Birmingham, Tech. Rep. 3, April 2015

[18] ARM, "Arm trustzone website," Online, 2014, accessed: 2014-06-24. [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone/index.php?tab=Hardware+Architecture

[19] Vasco, "Card readers," Online, 2016, accessed: 2016-05-19. [Online]. Available: https://www.vasco.com/products/two-factor-authenticators/hardware/card-readers/index.html

[20] G. S. Grewal, M. D. Ryan, L. Chen, and M. R. Clarkson, "Du-vote: Remote electronic voting with untrusted computers," in *28th Computer Security Foundations Symposium (CSF)*. IEEE, July 2015.

[21] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *Computer Security Foundations Symposium (CSF)*, vol. 25. IEEE, 2012.

[22] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGARCH Computer Architecture News*, 2012.

[23] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *USENIX Annual Technical Conference*, 2014.

[24] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, *Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?*, ser. Trust and Trustworthy Computing, CyLab, Carnegie Mellon University, USA. 2012.

[25] R. van Rijswijk-Deij and E. Poll, "Using trusted execution environments in two-factor authentication: comparing approaches," in *Open Identity Summit, OID*, 2013.

[26] D. Basin, S. Radomirovic, and M. Schläpfer, "A complete characterization of secure human-server communication," in *Computer Security Foundations Symposium (CSF), 2015 IEEE*.

[27] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *SIGOPS Operating Systems Review*, vol. 42. ACM, 2008.

[28] M. Kassner, "Keyscrambler: How keystroke encryption works to thwart keylogging threats," Online, October 2010, accessed: 2015-05-25. [Online]. Available: http://www.techrepublic.com/blog/it-security/keyscrambler-how-keystroke-encryption-works-to-thwart-keylogging-threats/

[29] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *Symposium on Security and Privacy (SP)*. IEEE, 2010.