Article

HyperIO: A Hypervisor-Based Framework for Secure IO

Michael Kiperberg [1,†] **and Nezer Jacob Zaidenberg** [2,3,*,†]

[1] Department of Software Engineering, Shamoon College of Engineering, Be'er Sheva 8410802, Israel
[2] School of Computer Sciences, Ariel University, Ariel 4076414, Israel
[3] Faculty of Information Technology, University of Jyväskylä, FI-40014 Jyväskylä, Finland
[*] Correspondence: nezerj@ariel.ac.il
[†] These authors contributed equally to this work.

Featured Application: We present a Firefox extension acting as a trust indicator and a clipboard encryptor. The framework can also be used for other means such as storing passwords, secure access to crypto wallet, etc.

Abstract: Malware often attempts to steal input and output through human interface devices to obtain confidential information. We propose to use a thin hypervisor, called "HyperIO", to realize a secure path between input and output devices using a partial implementation of device drivers. We apply our approach using two security systems built on HyperIO: FireSafe and ClipCrypt. FireSafe is a web browser extension which allows a remote web server to display and receive sensitive user information securely. ClipCrypt enables the user to securely enter and view their confidential information in commodity Windows applications.

Keywords: trusted path; virtualization; system security; browser security; malware protection

check for updates

Citation: Kiperberg, M.; Zaidenberg, N.J. HyperIO: A Hypervisor-Based Framework for Secure IO. *Appl. Sci.* **2023**, *13*, 5232. https://doi.org/10.3390/app13095232

Academic Editor: Vincent A. Cicirello

Received: 1 March 2023
Revised: 31 March 2023
Accepted: 12 April 2023
Published: 22 April 2023

1. Introduction

Confidential information, such as passwords, credit card numbers, and medical records, travel between the user's local machine and a remote server via secure protocols such as HTTPS [1]. Remote servers have excellent support from security specialists and have many protection layers, e.g., firewalls [2], antiviruses [3], and intrusion detection systems [4]. The user's local machine, however, is more vulnerable to attacks.

Keyloggers [5], screen capturing malware [6], and memory scrappers can fetch confidential information from the target process's memory, e.g., a web browser or memory of the kernel itself. Protecting personal information requires two building blocks:

- secure item communication with the IO devices;
- isolated computation environment.

ARM processors can realize these building blocks using ARM's TrustZone [7]. A software module executed in TrustZone's isolated environment, the "secure world", can prevent "normal world" software from accessing its internal data.

In addition, ARM's SoC devices belong to two categories: secure and non-secure [8]. The "NS bit" on the system bus tags each transaction's security. Transactions issued by secure devices set the NS bit to 0, whereas non-secure devices set this bit to 1. Secure devices will not handle transactions whose NS bit is 1. When the CPU issues a transaction, the current *world* determines the NS bit's value: a normal world sets the bit to 1 and a secure world to 0. The user can learn of the system state by a trust indicator.

On Intel CPUs, virtualization [9], and enclaves (SGX) [10] can achieve comparable memory isolation. However, Intel platforms lack built-in mechanisms for secure communication with external devices. Recently an attempt was presented [11] to introduce a secure communication channel between SGX enclaves and external devices. This approach relies on a thin hypervisor called "XMHF" [12]. Historically, BitVisor [13] was the first attempt to use virtualization for securing a hardware device. Specifically, BitVisor is a thin hypervisor that intercepts the communication between the operating system and a hard disk to implement full disk encryption transparently.

Following BitVisor's approach, we propose using a thin hypervisor, which we call "HyperIO", to secure users' sensitive information. HyperIO realizes a secure path between input and output devices by partially implementing device drivers by leveraging these devices' usually unused functionality. This approach allows us to achieve a small trusted code base (TCB) and a negligible performance overhead.

HyperIO intercepts the communication between the operating system and the keyboard. The keyboard's scroll lock LED [14] acts as a security indicator. HyperIO can block attempts to turn on the scroll lock LED from the operating system. In addition, HyperIO uses the display adapter's text mode to display sensitive information to the user. The hypervisor protects the memory region containing the displayed characters from malicious access. Because the text mode and associated memory region are not in use during normal execution of the operating system, HyperIO can freely utilize this resource to secure sensitive information while displaying it.

HyperIO is a framework for constructing systems with secure IO paths. We demonstrate the usefulness of HyperIO through two use cases. Both use cases demonstrate the ability to display and receive sensitive information from the user safely. The first use case is an extension for the Firefox web browser, which allows a remote web server to display information securely or receive sensitive user input. The second use case allows user-mode applications written for the Windows operating system to display and input sensitive information.

The main contributions of this paper are as follows:

- We present a novel thin hypervisor with a minimal TCB, which implements a secure IO path.
- We demonstrate our approach's applicability by describing the implementation of two security systems built using our hypervisor.
- We discuss the usability and security of HyperIO and compare it with previous works.
- We evaluate the performance of HyperIO and verify its effectiveness against popular keyloggers and screen-capturing software.

2. Background

2.1. Virtualization

Hardware-assisted virtualization [9] is a widely adopted foundation for security applications. Originally, virtualization extensions' main goals were to simplify virtual machine monitor design and improve performance. With these extensions, a virtual machine monitor, also called a "hypervisor", can configure the interception of various events inside the virtual machines (VMs). When an event occurs, the CPU transfers the control from the VM to the hypervisor. This transition is called a "VM exit". In addition, the CPU stores the information about the occurred event in a data structure for the hypervisor's latest inspection.

The hypervisor manages the memory of the VMs through a secondary-level address translation (SLAT) mechanism called "extended page tables" (EPT) by Intel. The hypervisor can define a page table for each VM, representing translation and access rights of the VM's physical addresses to real physical addresses. Using EPT, the hypervisor can isolate itself and the VMs from each other.

A hypervisor can intercept a wide range of events, while interception of some events, e.g., access to IO ports, can be disabled, others, e.g., execution of the CPUID instruction, induce VM exits unconditionally. We handle three types of events: (1) access to IO ports, (2) execution of the CPUID instruction, and (3) EPT violations. HyperIO and other hypervisors use the CPUID instruction as a hypercall instruction. The interception of access to IO ports and memory regions protects the sensitive IO and hypervisor.

2.2. TPM

The trusted platform module (TPM) provides secure, tamper-resistant storage and cryptographic functions that can operate on this storage. In addition to key storage functionality, symmetric and asymmetric cryptography, and hashing, TPM can seal a key to its internal state. Later the key can be fetched only when the TPM is in this exact state. The state of the TPM is defined by its platform configuration registers (PCRs). The PCRs can be read but not directly written. Instead, PCRs are extended by hashing their current value with some new value: $\mathrm{PCR}_n = \mathrm{hash}(\mathrm{PCR}_n, \mathrm{Value})$.

Each time an EFI application [15] is loaded, the EFI firmware extends the PCRs with the hash value of the application's image. When an EFI application seals a key, it guarantees that only the current and previously loaded applications can retrieve the key in the future. VirtSecIO, embedded in an EFI application, uses the TPM to store cryptographic keys that cannot be accessed by the operating system.

2.3. Certificates

HyperIO uses the X.509 public key certificates [16] in its communication. Each certificate contains the name and public key of a subject. The certificate itself is signed by a security authority. The verifier can verify the signature if it has the public key of the security authority. If not, the public key of the security authority can be sent in another certificate, signed by another security authority, thus creating a chain of certificates. The last certificate in the chain must be verifiable by a public key that is known to the verifier. These certificates are known as root certificates. All the web browsers are distributed with a set of pre-installed root certificates.

2.4. PS/2

The current implementation of HyperIO supports only PS/2 keyboards. We selected this type of keyboard for the simplicity of its communication protocol. Previous research [17] was motivated by this simplicity, too.

The communication with PS/2 keyboards is managed by a PS/2 controller [14]. The PS/2 controller uses two IO ports for its operation, 0x60 as a data port and 0x64 as a command and status port. Port 0x64 communicates with the controller, such as reading and writing its internal RAM and selecting the destination device. Port 0x60 sends and receives data from the previously selected destination device.

Assuming that the selected destination device is a PS/2 keyboard, reading from port 0x60 reads the keyboard's last scan code. A scan code represents a keyboard event, such as pressing or releasing a key. The scan codes can then be translated according to a conversion table. Some scan codes, e.g., presses and releases of shift keys, affect other scan codes' translation.

The act of writing to port 0x60 sends a command to the keyboard. The commands configure various aspects of a keyboard's behaviour. Some are single-byte; an argument follows others. For example, the 0xFF command is a single-byte command which resets the keyboard. The 0xF3 command defines, through its argument, the rate at which a pressed key produces events. Another example is the 0xED command, which configures the LEDs' state. The lower three bits in the byte that follows the 0xED command define the state of scroll lock (bit 0), number lock (bit 1), and caps lock (bit 2) LEDs.

2.5. Display Adapter

We use a simple and well-documented VMware SVGA II display adapter [18], provided by the VMware virtual machine monitor. VMware SVGA II provides three configuration resources: (a) IO registers, (b) frame buffer, and (c) memory queue. The IO registers to configure the display adapter. In particular, these registers allow the operating system to enable the SVGA mode, the frame buffer, and the memory queue. After this initialization, the operating system can request to redraw screen regions by queueing the appropriate requests. Of course, the data itself must reside in the frame buffer. This type of communication allows the operating system and the display adapter to work independently, thus improving the overall system performance.

The display adapter defines hundreds of internal registers, which can be accessed using two IO registers, "index" and "value". The operating system writes to an internal register x by setting the "index" register to x and writing the value to the "value" register. An internal register at index 1 controls the SVGA mode. When this register is zero, the SVGA mode is disabled, and the display adapter operates in the VGA mode. A text sub-mode can be selected in VGA mode, requiring the display adapter to interpret the data at physical address 0xB8000 as characters (and their attributes). When the SVGA mode is active, the display adapter does not use the memory region at the physical address 0xB8000.

3. System Design

3.1. Threat Model

We assume the attacker fully controls the local machine's software. Furthermore, the attacker has arbitrary code execution in kernel mode. Therefore, we impose only one limitation on the attacker: they cannot alter the firmware code and data. In particular, we assume that the code executing in the system management mode (SMM) and the EFI are trustworthy.

These assumptions are not very strong as most modern systems are equipped with TPM that verify the BIOS. A TPM is required to run Microsoft Windows™11.

3.2. System Architecture

Figure 1 depicts the architecture of HyperIO. HyperIO is a thin hypervisor embedded in an EFI application [15]. The EFI firmware loads the application, initializes the hypervisor, and then boots the operating system's bootloader, which executes inside the single virtual machine created by the hypervisor. Before initializing the hypervisor, the EFI application requests the EFI firmware to allocate a memory region for the hypervisor's code and data. This memory region is marked as reserved, and a benign operating system will not attempt to access it. However, to guarantee its security, the hypervisor uses the secondary-level address translation mechanism to make this memory region inaccessible to the VM. In addition, the hypervisor configures the IOMMU to ensure that this memory region is inaccessible from the peripheral devices. During the initialization, the hypervisor communicates with the TPM, retrieves the private key, and stores it in the secure memory region. HyperIO is able to acquire partial control over the screen and the keyboard. This control is sufficient for the HyperIO's operation on the one hand, and does not require HyperIO to include full drivers for these devices, thus keeping the code base at the required minimum.

HyperIO has two modes of operation: regular and secure, while in secure mode, HyperIO protects the keyboard's input and the screen's output. The keyboard's scroll lock LED acts as a security indicator; it is on if, and only if, the secure mode is active. HyperIO prevents all the operating system's attempts to modify the scroll lock LED state regardless of the operation mode. When HyperIO switches to the secure mode, the system behaviour changes as follows:

- HyperIO intercepts all the keystrokes and records them in a secure memory region;
- HyperIO switches the display adapter to the text mode; and
- HyperIO intercepts all attempts to change the video mode or access the display adapter's text mode buffer.

The main functionality provided by HyperIO is the secure input and output of sensitive information in response to a request performed by an external entity, which we call the "requester". The requester is identified by a certificate. The requester encrypts the output that it intends to display using HyperIO's public-key. HyperIO encrypts the user's input using the requester's public-key.

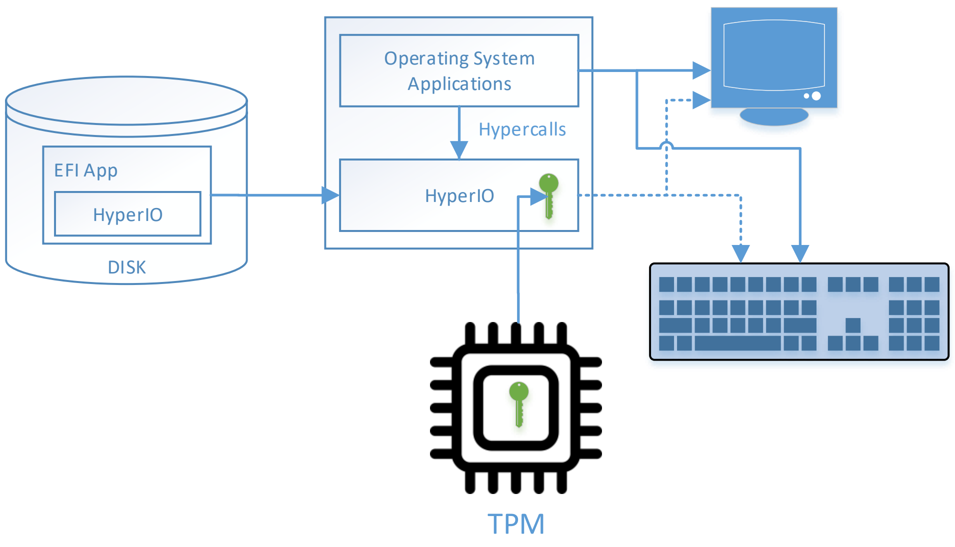


Figure 1. HyperIO's architecture. HyperIO is embedded in an EFI application, which is stored on a disk. During the boot, the EFI application is loaded, HyperIO is initialized and its private key is loaded from the TPM. HyperIO executes in an isolated execution environment, in which it can store its secret private key. HyperIO can obtain partial control over the screen and keyboard (depicted by dashed arrows), while the operating system has direct access to them (depicted by solid arrows). The operating system and the applications can communicate with the hypervisor via the exported hypercalls.

HyperIO exports its functionality via its two hypercalls: `HcDisplay` and `HcAcquire`. The `HcDisplay` hypercall receives two parameters: (a) an encrypted message to be displayed, (b) a certificate that identifies the requester. HyperIO reacts to this hypercall by switching to the secure mode and displaying the decrypted message and certificate. The hypercall returns immediately, thus allowing the operating system to handle interrupts and other events. HyperIO, however, remains in the secure mode until the user presses the "scroll lock" key. Until then, HyperIO replaces the intercepted keyboard strokes with an asterisk. An optional argument can be used to set another scancode for the replacement. The original keystrokes are translated to characters and recorded in a protected memory region, while the secure mode is active, HyperIO provides the users with a simple text editor, allowing them to scroll, navigate and edit the existing text.

The `HcAcquire` hypercall returns the recorded characters encrypted using the requester's public key. This hypercall receives a single parameter: a pointer to a buffer that will receive the encrypted data.

In the following sections, we describe the cryptographic infrastructure of HyperIO, the mode switching mechanism, and HyperIO's operation in each mode.

3.3. Initialization and Handshake

HyperIO uses public-key cryptography for sensitive data encryption and certificate verification. In particular, when sensitive data needs to be displayed, it is decrypted using HyperIO's private-key. Similarly, when sensitive information is input from the user, HyperIO encrypts it using the requester's public-key. HyperIO verifies the validity of the requester's public-key using the attached certificate.

HyperIO retrieves its private-key from the TPM during the initialization process and stores it in a secure memory region. To protect the secrecy of the private-key, HyperIO binds it to specific values of the PCR registers. The firmware updates the PCR registers with the hashes of the loaded EFI applications. Changing the boot order or the HyperIO's

EFI application's content will result in different PCR register values. Therefore, the retrieval of the private-key is possible only by the HyperIO's EFI application.

In the current implementation, the public-keys of the root certificate authorities are hard-coded in HyperIO. HyperIO implements RSA [19] for public-key cryptography and AES [20] for symmetric cryptography. We assume that the public-key of HyperIO is known to a remote server. However, this assumption can be relaxed by distributing HyperIO's public-key with a certificate that proves its authenticity.

The encryption scheme is similar to TLS [21]. HyperIO encrypts the sensitive information using AES and a randomized key. The key itself is encrypted using RSA.

Assume that the requester wishes to display the secret message m. The requester performs the following steps:

1. Generates a random number r.
2. Computes $c_1 \leftarrow \text{AES}_r(m)$.
3. Computes $c_2 \leftarrow \text{RSA}_{pk_H}(c_1)$, where pk_H is HyperIO's public key.
4. Computes $s \leftarrow \text{RSA-Sign}_{sk_R}(c_1||c_2)$, where sk_R is the requester's private key.
5. Computes the full payload $p \leftarrow \text{cert}_1||...||\text{cert}_k||s||c_1||c_2$ where cert_k is the requester's certificate signed by cert_{k-1} and so on, and cert_1 is a certificate hard-coded in HyperIO.
6. Performs `HcDisplay`(p)

When HyperIO receives the request, it validates the chain of certificates, decrypts using its private key, the random key r and then decrypts the message m. If an error occurs in one of the decryption/validation steps, HyperIO terminates the hypercall and returns to the guest.

If the requester wishes to input sensitive information, HyperIO encrypts the input using a random key r and AES, encrypts r using the requester's public key and RSA, signs everything using HyperIO's private key and RSA and sends the result back to the requester.

3.4. Implementation Details

HyperIO is based on a thin hypervisor, which intercepts a minimal set of events. The hypervisor does not emulate the peripheral devices and therefore does not support multiple operating systems' execution. These properties allow the hypervisor to incur an insignificant performance overhead. The hypervisor intercepts only events that are required to achieve its goals. Some events are intercepted only in secure mode; others are intercepted in the regular mode. We can divide the intercepted events into two categories: (a) access to IO ports and (b) access to memory regions. HyperIO intercepts access to IO ports by setting the corresponding bits in the IO-bitmap of the VMCS. Access to memory regions are intercepted by zeroing the EPT entries' access rights.

HyperIO assumes that the system is equipped with a PS/2 keyboard and monitors port 0x60, the IO port that the operating system uses for communication with the keyboard. Writing to port 0x60 is intercepted in both modes of operation. The operating system configures the keyboard LEDs by writing a sequence of two bytes to port 0x60. The first byte is 0xED, and the second byte defines the state of the LEDs. Specifically, the scroll lock LED's state is defined by the second byte's least significant bit. HyperIO sets this bit according to the current operation mode, regardless of the operating system's setting.

When the user presses or releases a key, the keyboard generates an interrupt. The operating system reacts to this interrupt by reading from port 0x60. When in the secure mode, HyperIO intercepts the operating system's attempt to read from port 0x60 and replaces the actual code of the pressed key with the code of the asterisk key.

During the transition into the secure mode, HyperIO gains control of the display adapter in two steps:

1. the display mode is selected to be text mode;
2. the memory region that controls the displayed characters is configured to be inaccessible.

In text mode, the display adapter uses the memory region that resides at physical address 0xB8000. HyperIO puts in this region the text that it was requested to display. In video mode, the display adapter uses a different memory region configured by the operating system. Therefore, the operating system can freely modify the video mode memory region while the secure mode is active without affecting the pixels that are displayed on the screen, while in secure mode, HyperIO prevents malicious modification of the display adapter's video mode by intercepting the operating system's access to the configuration registers. The PCI configuration space reports the location of these ports. Typically, the operating system accesses the configuration registers of the display adapter only during its initialization. We can assume that a benign operating system never accesses the memory regions secured by the hypervisor after the operating system's initialization completes. Therefore, in our current implementation, HyperIO resets the PC when malware attempts to access these memory regions.

4. Related Work

The secure input and output solutions vary by their security foundation. For example, SwitchMan [22] is a user-mode framework which enables a remote server to request secure input and output. The central claim is that a remote server should determine various fields' sensitivity in a web application. FireSafe follows this idea but implements it using a secure hypervisor without including an entire operating system in the TCB. This improvement in security does not sacrifice usability. Similarly to SwitchMan, FireSafe performs automatic switches between the secure and regular modes. The only deficiency of FireSafe compared to SwitchMan is minor performance degradation due to the presence of a hypervisor.

Other works provide more robust security guarantees by leveraging an external device or processor's security features, such as system management mode (SMM), virtualization [9], and software guard extensions (SGX) [10]. We discuss these in the following sections.

4.1. External Device

Fidelius [23], and its successor ProtectIon [24], provide the secure input and output of sensitive information, such as credential fields in web forms. Fidelius uses enclaves, which implement the network (web) protocols, and a secure external device to accomplish its tasks. The device acts as a mediator between the computer and the peripheral devices. It captures keystrokes and modifies the monitor's image. When sensitive information is required, the external device switches to a secure mode and transmits the keystrokes in an encrypted form to the enclave. Moreover, the browser renders the web form on the image sent to the monitor. Fidelius is superior to HyperIO in terms of usability because the sensitive information is displayed graphically in its natural position on the screen. In addition, Fidelius has better performance because the external device performs the additional computations. However, because HyperIO does not require an external device, its deployment is easier and cheaper.

Bumpy [25] and BitE [26] provide secure input and output by leveraging Flicker [27]. Bumpy makes two assumptions regarding the input and output devices. Firstly, Bumpy assumes that the input devices can encrypt the user's input for later decryption by Bumpy's PALs protected by Flicker. Secondly, Bumpy assumes an additional external monitor is connected to the computer. This monitor is used to display sensitive information by Bumpy's PALs. Unlike Bumpy, HyperIO does not require special devices for its secure operation.

4.2. SMM

Aurora [28] leverages SMM to provide secure channels between enclaves and devices. Because the code executed in SMM is part of the firmware, any new code must be signed and deployed by the computer's manufacturer. HyperIO, on the other hand, can be deployed on ordinary computers without cooperation with the manufacturer. Furthermore, unlike HyperIO, Aurora does not experience a constant system slow-down due to the presence of a hypervisor.

TrustLogin [17] uses SMM as an isolated environment, and similarly to HyperIO, it uses the keyboard LEDs as security indicators. The main goal of TrustLogin is to secure the input of credentials. However, unlike HyperIO, it does not provide a means for a secure output. TrustLogin is comparable to FireSafe, a browser extension which provides secure input and output. Both FireSafe and TrustLogin provide the convenient and secure input of sensitive information. In addition, FireSafe provides a secure output and the ability to edit previously entered (and encrypted) information.

4.3. Hypervisor

T-PIM [29] uses a full hypervisor to run two virtual machines. One virtual machine is a general workstation, whereas the second provides a secure input. T-PIM automatically switches between the two virtual machines when a sensitive input is required. Compared to T-PIM, HyperIO, based on a thin hypervisor, has a much smaller TCB and better performance.

Qubes OS [30] uses a full hypervisor (Xen [31]) to run multiple virtual machines. The displays of all the virtual machines are combined such that the system behaves as expected from the user's perspective. The frame of each window indicates the virtual machine which hosts the window's application, while Qubes OS improves overall security, a vulnerability in the kernel can disclose the compromised virtual machine's sensitive information. Furthermore, similarly to T-PIM, Qubes OS has a large TCB and non-negligible performance overhead.

SGXIO [11] uses a hypervisor to provide a secure input and output for SGX enclaves. The hypervisor includes drivers for devices whose input or output need to be protected. Furthermore, the hypervisor emulates two virtual devices for each such device: secure and non-secure. The operating system can access the non-secure device, whereas an SGX enclave can only access the secure device. The multiplexing between the two devices can be temporal, i.e., at different times, the actual device is owned exclusively by the operating system or the enclave. It can also be spatial, i.e., the devices' existing resources are divided between the operating system and the enclave. SGXIO provides a generic solution for the trusted IO path problem, and its ideas can be used to implement HyperIO as an enclave. However, the device emulation harms both the simplicity and performance of systems based on SGXIO. The design of HyperIO is much simpler, thus allowing it to have a small TCB and achieve a minor performance penalty.

Zhou et al. [32] described a generic method to build a trusted IO path between a device and a user application, using a thin hypervisor as a trusted codebase and isolation provider for the sensitive IO. Specifically, the authors discussed the applicability of their design to USB devices. Using these ideas, HyperIO can be extended to support not only PS/2 keyboards but also USB keyboards.

In our prior work, HyperPass [33], we described a hypervisor-based method that allows the user to enter their password securely. Unlike HyperIO, HyperPass is not a generic framework. HyperPass can be considered an application of HyperIO, similar to FireSafe and ClipCrypt. However, some of the ideas of HyperPass are implemented in HyperIO.

5. Discussion

This section evaluates HyperIO from three perspectives: security, convenience, and performance. Finally, we compare HyperIO security to its competitors in Table 1 in and outline the benefits and drawbacks of our approach.

5.1. Security

To assess the security of HyperIO's design and the correctness of its implementation, we used several popular programs for screen capturing [34] and keylogging [35], see Table 2. With each program, we performed a short test, which included:

1. entering the secure mode;

2. typing a short message;
3. exiting the secure mode.

In all cases, the keyloggers were unable to record the keystrokes. However, the screen-capturing programs continued to capture the operating system's video mode output.

Table 1. Trusted code base (TCB) of related systems. The second column should be interpreted as follows: OS = operating system; FW = firmware; FHV = full hypervisor; and THV = thin hypervisor.

Solution	Trusted Code Base	External Device
SwitchMan	OS + FW	–
Fidelius/ProtectIon	–	V
Bumpy/BitE	–	V
Aurora	FW	–
TrustLogin	FW	–
T-PIM	FHV + FW	–
Qubes OS	FHV + FW	–
SGXIO	Drivers + THV + FW	–
HyperIO	THV + FW	–

Table 2. Screen-capturing and keylogging programs.

Screen Capturing	Key Logging
OBS Studio	Spyrix Free Keylogger
FlashBack Express	Actual Keylogger
Debut Video Capture and	All In One Keylogger
ShareX	HomeGuard Activity Monitor

We measure the security of HyperIO and its competitors in terms of size (less SLOC is better). Table 3 presents the trusted code base of HyperIO and its competitors, while Fidelius, ProtectIon, Bumpy, and BitE have (almost) no software components, they require an external device to operate. Aurora and TrustLogin use the system management mode (SMM), which executes the firmware code. They have the smallest TCB among the systems that do not require external devices because all other systems must also trust the firmware code, which executes in the most privileged mode. The TCBs of T-PIM and Qubes OS include a full hypervisor. The size of a full hypervisor, such as Xen, is ≈600,000 SLOC. In Qubes OS, the sensitive information's safety also depends on the set of applications that share the same AppVM.

Similarly to HyperIO, SGXIO is a thin hypervisor with a small TCB. However, SGXIO includes a full implementation of drivers for devices that act as sources or sinks of sensitive information. The drivers of display adapters are known for their large size. For example, the driver of a simple driver for the VMware SVGA II has ≈33,000 SLOC [36]. The code that handles the display adapter in HyperIO has ≈100 SLOC. HyperIO's hypervisor has ≈2000 SLOC, the cryptography library has ≈500 SLOC, and the editing code has ≈500 SLOC, summing up to ≈3000 SLOC. Furthermore, SGXIO requires SGX support. SGX is unavailable on AMD (SEV is the proposed alternative) and older CPUs (core i3/5/7 processors from 1st to 5th generations) by Intel. Furthermore SGX is now deprecated [37] and no longer available on newer Intel CPUs. (core i3/5/7 processors generations 11th and above). SGX technology imposes many complications when delivering information into and out of a trusted enclave. The technology was not massively adopted by the industry and is no longer available in the three latest Intel CPU generations. SGX imposes many limitations on SGXIO requiring significant work on drivers not required in HyperIO and has a much larger footprint.

5.2. Performance

HyperIO uses a thin hypervisor, which may degrade the overall system performance. To assess this performance degradation, we ran a benchmarking tool called PCMark [38] in four configurations:

1. without a hypervisor;
2. with HyperIO's hypervisor;
3. with HyperIO (which includes additional interceptions, e.g., key presses);
4. with Oracle VirtualBox (6.1.14) as an example of a full hypervisor, running the same version of Windows with 4GB of RAM.

Table 3 summarizes the exact configuration of our testing environment. Table 4 presents the overhead percentage of each configuration compared to the base configuration without a hypervisor.

Table 3. Testing environment configuration.

Host CPU	Intel(R) Core(TM) i7-10610U
Host memory	16 GB
Host OS	Ubuntu 20.04.1 LTS
VMM	VMware Workstation 15.5.6
Guest CPU	Intel(R) Core(TM) i7-10610U
Guest memory	8 GB
Guest OS	Windows 10 (19041)

Table 4. Performance degradation in percentage concerning the configuration without a hypervisor.

Category	**Thin Hypervisor**	**HyperIO**	**VirtualBox**
App start-up	2.29	6.01	44.92
Video conferencing	−8.36	−8.02	43.78
Web browsing	2.66	2.75	32.75
Spreadsheet	1.18	2.62	39.91
Writing	1.54	2.69	42.66
Photo editing	−0.24	0.38	38.35
Video editing	1.77	5.7	28.02

A thin hypervisor and HyperIO incur a minor performance degradation (≈1.35% and ≈2.88% on average, even without considering the video conferencing results). The only difference between the thin hypervisor and HyperIO is the interception of the input and output on port 0x60. Because the operating system accesses this port only upon an interrupt from the keyboard, a relatively rare event, our tests show a minor performance degradation due to this interception. A full hypervisor, represented by Oracle VirtualBox, showed a much higher impact on the overall system performance (≈38.6% on average), while there is clear evidence that HyperIO affects the overall system performance, the impact is sufficiently low to allow its installation even in CPU-intensive workstations.

6. Applications

We demonstrate the usefulness of HyperIO by presenting two use cases. Both use cases demonstrate the ability to display or receive sensitive information without making it vulnerable to theft by malware.

These applications provide an example to the main use cases of the HyperIO environment.

1. Trustworthy network paths;
2. Trustworthy human interface devices;
3. Trustworthy screen output.

The first use case is an extension for the Firefox web browser, which allows a remote webserver to securely input and output sensitive user information. This application demonstrates secure network channel and trustworthy network interfaces as well as trustworthy human interface devices (keyboard). The second use case allows user-mode applications written for the Windows operating system to display sensitive information or receive sensitive input from the user. This application demonstrate trustworthy screen output and trustworthy human interface devices. In the following sections, we describe the implementation of these two applications.

6.1. FireSafe—Secure IO in Web Applications

Web applications often display sensitive data, such as medical records, clients' private information, and the contents of private messages. Even more often, web applications request the user to input sensitive information, such as passwords and credit card numbers. This section presents an extension to the Firefox web browser, enabling a remote server to request the secure input and output of sensitive information. We call our extension "FireSafe".

FireSafe acts as a mediator between the server and HyperIO by issuing hypercalls in response to various events occurring on the HTML page sent by the server. The server submits its request for secure input and output by embedding a "data-hyper-io" attribute in an HTML tag. When an event occurs in such a tag, FireSafe issues a hypercall. Since web browsers ignore the "data-*" attributes, the web server can send the same HTML page to a FireSafe-enabled web browser and a regular web browser. The page will be displayed correctly, but without FireSafe, the user cannot display the sensitive information sent by the server.

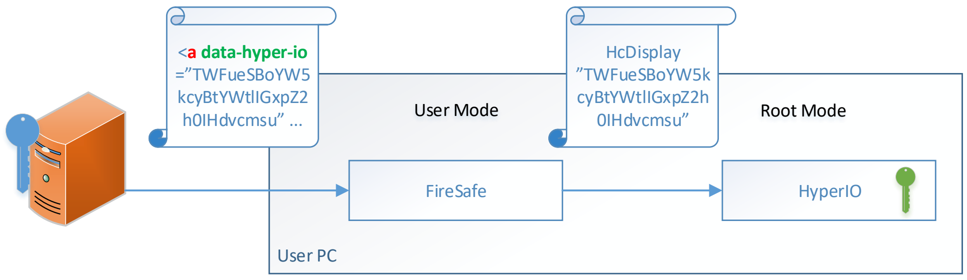
In the current implementation, the set of tags supported by FireSafe is limited. The server embeds the "data-hyper-io" attribute in an "A" tag (the A tag usually represents a hyperlink) to display sensitive information. The content of this tag can be text or an image. When a user clicks on the tag, FireSafe issues the `HcDisplay` hypercall. The values of the parameters for this hypercall are encoded using Base64 in the "data-hyper-io" attribute. The "href" attribute should contain an empty string to prevent redirection to another URL.

The following example demonstrates the simplicity of the proposed system from the user's point of view (see Figure 2). Consider a medical web portal containing the user's private medical records. Usually, the portal includes several sub-pages. One of these pages list, for example, the results of the user's blood tests ordered by date. The user can view the results by clicking on the test title. The test title is an "A" tag that redirects the user to the test results page. With FireSafe and HyperIO, the server adds the "data-hyper-io" attribute to the "A" tag and clears its "href" attribute. The "data-hyper-io" attribute contains the test results in an encrypted form. After clicking on the "A" tag, FireSafe issues a hypercall. This hypercall requests HyperIO to switch to the secure mode and display the blood test results. From the user's perspective, the system behaves similarly to the standard case, with the only difference being the transition of the display adapter to text mode. The user exits the text mode by pressing the "scroll lock" key.

To request sensitive input, the server embeds the "data-hyper-io" attribute in an "INPUT" tag, representing a text or password field (see Figure 3). When the tag becomes focused, FireSafe issues the `HcDisplay` hypercall. Similarly to the "A" tag, the parameters for this hypercall are encoded using Base64 in the "data-hyper-io" attribute. The hypercall returns immediately, but the display adapter remains in text mode until the user presses the "scroll lock" key. Next, all the characters the user enters are replaced by "*". Then, the "scroll lock" key is replaced by "!". FireSafe monitors the value of the "INPUT" tag by registering to its "change" event. In response to "!", FireSafe issues the `HcAcquire` hypercall, which returns the user's input in its encrypted form. The input is then converted to Base64 and set as the value of the "INPUT" tag. Upon submitting the form containing the "INPUT" tag, the browser sends the encrypted user input to the server.

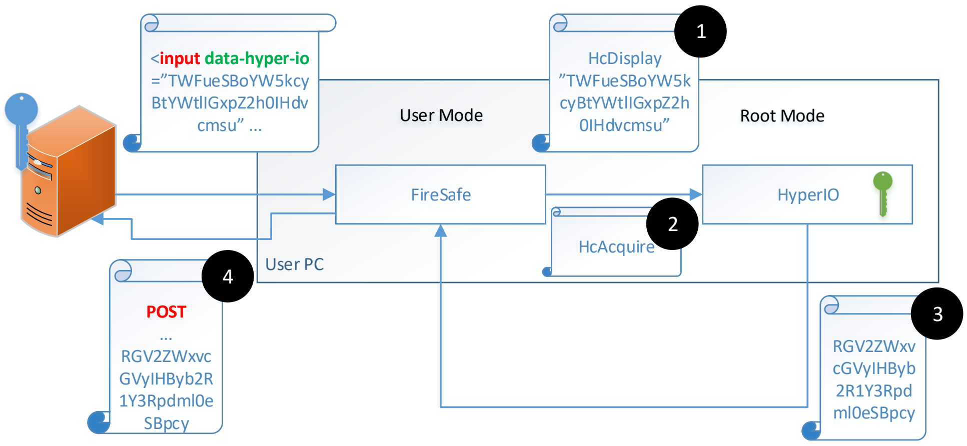
We demonstrate the simplicity of the proposed secure user input scheme using the previous example of the medical web portal. First, the portal requests to enter the user's login name and password to log in. The server embeds the "data-hyper-io" attribute in the password field (which is represented by the "INPUT" tag). When the password field receives focus due to a click, FireSafe issues the `HcDisplay` hypercall, thus requesting HyperIO to switch to the secure mode. HyperIO displays a prompt message (the "data-hyper-io" attribute has the prompt message). Next, the user enters their password and presses the "scroll lock" key. FireSafe reacts by issuing the `HcAcquire` hypercall and setting the password field's value to the encrypted password. Finally, the user clicks the "login" button to submit the form to the server. As in the previous example, the user experience is similar to the standard case.

FireSafe demonstrates that a remote server can use HyperIO's functionality to securely input and display sensitive information. Moreover, from the user's perspective, the system preserves its natural behaviour.



<a data-hyper-io ="TWFueSBoYW5kcyBtYWtlIGxpZ2h0IHdvcmsu" ...
HcDisplay "TWFueSBoYW5kcyBtYWtlIGxpZ2h0IHdvcmsu"
User Mode
Root Mode
FireSafe
HyperIO
User PC

Figure 2. Webserver's request to display sensitive information on the user's screen. The "A" tag embeds the "data-hyper-io" attribute. Upon clicking the tag, FireSafe issues the `HcDisplay` hypercall to HyperIO. HyperIO uses its private key to decrypt the sensitive information, switches to the secure mode, and displays the decrypted information.



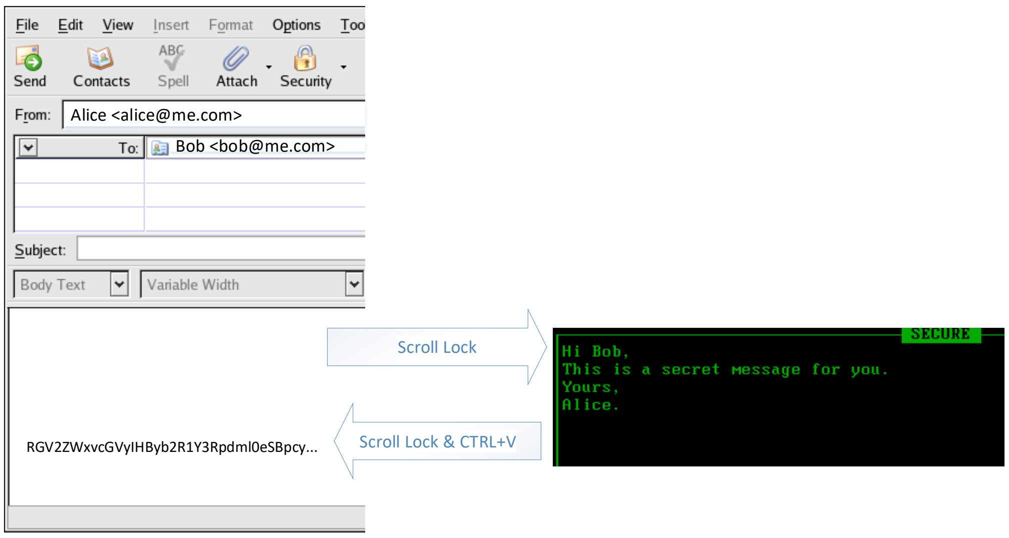
<input data-hyper-io ="TWFueSBoYW5kcyBtYWtlIGxpZ2h0IHdvcmsu" ...
HcDisplay "TWFueSBoYW5kcyBtYWtlIGxpZ2h0IHdvcmsu"
1
User Mode
Root Mode
FireSafe
HyperIO
2
HcAcquire
User PC
4
POST ... RGV2ZWxvcGVyIHByb2R1Y3Rpdml0eSBpcy
3
RGV2ZWxvcGVyIHByb2R1Y3Rpdml0eSBpcy

Figure 3. Webserver's request to input sensitive information from the user's keyboard. The "input" tag embeds the "data-hyper-io" attribute. Upon clicking the input field, FireSafe issues the `HcDisplay` hypercall (step 1) followed by the `HcAcquire` hypercall (step 2). HyperIO uses its private key to decrypt the sensitive information transmitted by the `HcDisplay` hypercall, switches to the secure mode, and displays the decrypted information. Then HyperIO inputs the sensitive information from the keyboard, encrypts it using the requester's public key, and returns it to FireSafe (step 3), which transmits it to the server in the following POST HTTP request (step 4).

6.2. ClipCrypt—Secure IO in Graphical Applications

Regular user-mode applications, such as email viewers, text editors, and database managers, often display sensitive information on the screen, making this information vulnerable to screen scrapers and other types of malicious software. We present ClipCrypt, a clipboard encryption system for the Windows operating system based on HyperIO. ClipCrypt allows the user to display and edit the sensitive information residing in the clipboard. To display previously encrypted sensitive data, the user places it in the clipboard (for example, by selecting the text and choosing copy in the context menu) and presses the "scroll lock" key. In response, ClipCrypt issues the `HcDisplay` hypercall, passing the clipboard's content and HyperIO's certificate as parameters. ClipCrypt sets the optional argument of `HcDisplay` to set the replacement scancode to the "scroll lock" key. HyperIO switches to the secure mode, decrypts the sensitive information using its private key, and displays it to the user. The user views or edits the text. HyperIO replaces all the user's input with the "scroll lock" key, thus not affecting the operating system and its applications. When the user completes editing, they press the "scroll lock" key. In response, HyperIO switches back to the regular mode. ClipCrypt issues the `HcAcquire` hypercall, which causes HyperIO to return the user's sensitive information encrypted using HyperIO's public key. Afterward, ClipCrypt stores the encrypted data in the clipboard. ClipCrypt converts the data it retrieves and stores it in the clipboard from and to the Base64 format.

From the user's perspective, pressing the "scroll lock" key invokes the encryption/decryption functionality. Otherwise, the behaviour of the system remains identical to the standard case. However, this approach is sufficiently general to be applied to various use cases. For example, suppose Alice wants to send a confidential email to Bob. In this case, she can open her favourite email client, press the "scroll lock" key, enter the body of the email, press the "scroll lock" key again, and paste the encrypted body by pressing "CTRL + V". After receiving the email, Bob can view it by selecting the encrypted body, pressing "CTRL + C" to place it in the clipboard, and finally pressing the "scroll lock" key to decrypt and display it on the screen. This secure transmission method from Alice's keyboard to Bob's display is not limited to sending emails. It allows the user to write and view sensitive documents, database entries, chat messages, etc. Figure 4 illustrates the ClipCrypt's operation in this scenario.



Alice <alice@me.com>

Bob <bob@me.com>

RGV2ZWxvcGVyIHByb2R1Y3RpdmI0eSBpcy...

Hi Bob,
This is a secret message for you.
Yours,
Alice.

Figure 4. Using ClipCrypt to encrypt the body of an email. The user presses the "scroll lock" key to switch to the secure mode. In the secure mode, the user can edit the text they wish to encrypt. Then, the user presses the "scroll lock" key again. The encrypted message is stored in the Windows clipboard. By pressing the "CTRL + V" key combination, the user can paste the encrypted message into the body of the email.

ClipCrypt consists of a window-less user-mode application and a kernel-mode driver, the Interception API project [39]. This allows our user-mode application to register for various keyboard events. Specifically, the user-mode application handle releases of keyboard keys.

When the user presses the "scroll lock" key, ClipCrypt fetches the clipboard content's textual representation. Whereas the clipboard may contain multiple formats of the same data simultaneously, e.g., formatted text, regular text, and image, HyperIO's output is limited to a plain textual representation. ClipCrypt translates the clipboard's textual representation from Base64 to binary and passes it as the first parameter to the `HcDisplay` hypercall. The second parameter is the certificate of HyperIO, which will instruct HyperIO to encrypt the user's input using its public key. If the clipboard is empty, ClipCrypt transmits to HyperIO an encrypted empty string. This behaviour allows the user to start writing an encrypted message from scratch.

HyperIO switches back to the regular mode when the user presses the "scroll lock" key, which is replaced by "!". ClipCrypt responds to "!" by issuing the `HcAcquire` hypercall. Then, ClipCrypt translates the encrypted input to the Base64 format and copies it to the clipboard.

Although our approach is sufficiently general to be applied in various circumstances, it suffers from two deficiencies. The first is a slight inconvenience. Consider, for example, a situation where the user wants to edit a text field containing sensitive information. Using ClipCrypt, the user must perform the following steps:

1. select the current content by pressing "CTRL + A";
2. copy this content to the clipboard by pressing "CTRL + C";
3. press the "scroll lock" key to enter the secure mode;
4. perform the actual editing and presses the "scroll lock" key to return to the regular mode;
5. paste the content of the clipboard by pressing "CTRL + V".

Step 4, in which the user performs the editing, cannot be discarded. Other steps, however, can be automated by ClipCrypt, as explained below. The idea is to emulate key presses using the keybd_event function of the Win32 API to perform selection, copying and pasting in the focused component. ClipCrypt performs the emulation when the user clicks an editable component. Due to the great variety of editable components, ClipCrypt allows the user to configure the set of components that are considered editable. We have successfully implemented this approach in programs distributed by Microsoft: Word, Outlook, and Notepad, and some non-Microsoft programs, such as the mIRC chat program and Notepad++. Unfortunately, some applications, including LibreOffice Writer and Mozilla ThunderBird, do not use Windows API for rendering their components. From ClipCrypt's point of view, these applications resemble a single component and cannot be supported.

Upon clicking an editable component, ClipCrypt invokes the keybd_event function (multiple times) to select and copy the first nine characters that follow the input cursor to the clipboard. For example, if the copied string is the magic "CLIPCRYPT," ClipCrypt will request HyperIO to switch to the secure mode. ClipCrypt copies the following eight characters, representing the encrypted data's length in a hexadecimal format. Then, ClipCrypt copies the encrypted data and issues the `HcDisplay` hypercall. Finally, ClipCrypt prepends the encrypted data returned by the `HcAcquire` hypercall with its length and a magic string to support this additional functionality.

The second drawback is making the user responsible for classifying specific inputs as sensitive. The end-user may need to be qualified to decide whether particular information is sensitive and should be encrypted. Moving this responsibility from the end user to a security specialist may be beneficial. An application that requires the input or output of sensitive information can be adapted to directly issue the hypercalls of HyperIO. A less challenging solution can be applied in "document editors". Notable examples of such "editors" are Microsoft Word, Microsoft Outlook, Notepad, and virtually any other application that allows one to save and open documents. Security specialists can define templates with pre-encrypted text of zero length for such applications. After opening the template and clicking on the encrypted text, the system automatically switches to secure mode.

6.3. Convenience

We consider solutions that rely on an external device, such as Fidelius, ProtectIon, Bumpy, and BitE, inconvenient due to the device's additional cost. However, this consideration may not hold when this cost is negligible compared to the project's overall budget.

Qubes OS and T-PIM require the user to transition to a secure environment. In Qubes OS, the user must enter sensitive information in the correct AppVM. In T-PIM, the user must switch between the regular and trusted VM to enter their password. HyperIO and the two applications built on top of it provide an automatic transition to the secure mode.

TrustLogin allows the user to enter their password "in place". Unlike HyperIO, TrustLogin does not allow users to view and edit the currently entered string. Furthermore, unlike FireSafe, TrustLogin does not provide a means to send sensitive information from a remote server to the user.

The main goal of SwitchMan is convenience. As such, SwitchMan provides an automatic transition to the secure mode and back by a remote request. FireSafe provides similar automation; however, the main drawback of FireSafe is its textual output.

Another limitation of HyperIO is its support for only PS/2 keyboards. However, Zhou et al. [32] presented an approach that HyperIO can adopt to support USB and Bluetooth keyboards.

7. Conclusions

HyperIO provides an infrastructure for the secure input and output of sensitive information. This infrastructure can be used in the realization of security applications. We demonstrated the usefulness of HyperIO's approach using two applications: FireSafe, a Firefox extension, which allows a remote server to input and output a user's sensitive data, and ClipCrypt, a clipboard encryption service for the Windows OS. HyperIO incurs a minor performance degradation, making it suitable for deployment even on CPU-intensive workstations.

Author Contributions: Conceptualization, M.K. and N.J.Z. ; Methodology, M.K. and N.J.Z.; software, M.K. validation, M.K.; investigation, N.J.Z.; writing—original draft preparation, M.K.; writing—review and editing, N.J.Z.; project administration, N.J.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Please contact the correspondence author for data and source.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

FHV	full hypervisor
FW	firmware
IO	Input/output
OS	operating system
SGX	Secure guard extentions
SLOC	Software lines of code
SMM	System management mode
THV	thin hypervisor
TPM	Trusted platform module
VMM	Virtual machine monitor

References

1. Rescorla, E. RFC2818: Http over TLS, 2000. Available online: https://dl.acm.org/doi/abs/10.17487/RFC2818 (accessed on 16 February 2023).
2. Sharma, R.K.; Kalita, H.K.; Issac, B. Different firewall techniques: A survey. In Proceedings of the Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Hefei, China, 11–13 July 2014 ; pp. 1–6.
3. Gandotra, E.; Bansal, D.; Sofat, S. Malware analysis and classification: A survey. *J. Inf. Secur.* **2014**, *2014*. [CrossRef]
4. Axelsson, S. *Intrusion Detection Systems: A Survey and Taxonomy*; Technical Report; 2000. Available online: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7a15948bdcb530e2c1deedd8d22dd9b54788a634 (accessed on 16 February 2023).
5. Hussain, M.; Al-Haiqi, A.; Zaidan, A.; Zaidan, B.; Kiah, M.M.; Anuar, N.B.; Abdulnabi, M. The rise of keyloggers on smartphones: A survey and insight into motion-based tap inference attacks. *Pervasive Mob. Comput.* **2016**, *25*, 1–25. [CrossRef]
6. Farinholt, B.; Rezaeirad, M.; McCoy, D.; Levchenko, K. Dark Matter: Uncovering the DarkComet RAT Ecosystem. In Proceedings of the Web Conference 2020, Online, 20 April 2020; pp. 2109–2120.
7. Pinto, S.; Santos, N. Demystifying ARM TrustZone: A comprehensive survey. *ACM Comput. Surv. (CSUR)* **2019**, *51*, 1–36. [CrossRef]
8. Rosenbaum, A.; Biham, E.; Bitan, S. Trusted Execution Environments. Ph.D. Thesis, Computer Science Department, Technion, Haifa, Israel, 2019.
9. Neiger, G.; Santoni, A.; Leung, F.; Rodgers, D.; Uhlig, R. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technol. J.* **2006**, *10*, 167. [CrossRef]
10. Costan, V.; Devadas, S. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* **2016**, *2016*, 1–118.
11. Weiser, S.; Werner, M. Sgxio: Generic trusted i/o path for intel sgx. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 261–268.
12. Vasudevan, A.; Chaki, S.; Jia, L.; McCune, J.; Newsome, J.; Datta, A. Design, implementation and verification of an extensible and modular hypervisor framework. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 430–444.
13. Shinagawa, T.; Eiraku, H.; Tanimoto, K.; Omote, K.; Hasegawa, S.; Horie, T.; Hirano, M.; Kourai, K.; Oyama, Y.; Kawai, E.; et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Washington, DC, USA, 11–13 March 2009; pp. 121–130.
14. Chapweske, A. The PS/2 Mouse/Keyboard Protocol. 2003. Available online: http://www.computer-engineering.org/ps2protocol (accessed on 16 February 2023).
15. Zimmer, V.; Rothman, M.; Hale, B. UEFI: From Reset Vector to Operating System. In *Chapter 3 of Hardware-Dependent Software*; Springer: Berlin/Heidelberg, Germany, 2009.
16. Gerck, E.; et al. Overview of Certification Systems: x. 509, CA, PGP and SKIP. *Black Hat Briefings* **1997**, *99*. Available online: https://www.researchgate.net/profile/Ed-Gerck/publication/318700731_original_web_site_Overview_of_Certification_Systems/data/597829520f7e9b277721d8ce/certhtm.pdf (accessed on 16 February 2023)
17. Zhang, F.; Leach, K.; Wang, H.; Stavrou, A. Trustlogin: Securing password-login on commodity operating systems. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14–17 April 2015; pp. 333–344.
18. Dowty, M.; Sugerman, J. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Oper. Syst. Rev.* **2009**, *43*, 73–82. [CrossRef]
19. Milanov, E. *The RSA algorithm*; RSA Laboratories: Hebron, Connecticut, 2009; pp. 1–11.
20. Daemen, J.; Rijmen, V. Reijndael: The Advanced Encryption Standard. *Dr. Dobb's J. Softw. Tools Prof. Program.* **2001**, *26*, 137–139.
21. Thomas, S. *SSL and TLS Essentials*; Wiley: New York, NY, USA, 2000; Volume 3.
22. Zheng, S.; Zhou, Z.; Tang, H.; Yang, X. SwitchMan: An Easy-to-Use Approach to Secure User Input and Output. In Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 19–23 May 2019; pp. 105–113.
23. Eskandarian, S.; Cogan, J.; Birnbaum, S.; Brandon, P.C.W.; Franke, D.; Fraser, F.; Garcia, G.; Gong, E.; Nguyen, H.T.; Sethi, T.K.; et al. Fidelius: Protecting user secrets from compromised browsers. *IEEE Symp. Secur. Priv.* **2019**, *1*, 264–280.
24. Dhar, A.; Ulqinaku, E.; Kostiainen, K.; Capkun, S. ProtectIOn: Root-of-Trust for IO in Compromised Platforms. *IACR Cryptol. ePrint Arch.* **2019**, *2019*, 869.
25. Perrig, J.M.M.A.; Reiter, M.K. Safe passage for passwords and other sensitive data. In Proceedings of the 16th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 8–11 February 2009.
26. McCune, J.M.; Perrig, A.; Reiter, M.K. Bump in the Ether: A Framework for Securing Sensitive User Input. In Proceedings of the USENIX Annual Technical Conference, General Track, Boston, MA, USA, 30 May–3 June 2006; pp. 185–198.
27. McCune, J.M.; Parno, B.J.; Perrig, A.; Reiter, M.K.; Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems, Glasgow, UK, 1–4 April 2008; pp. 315–328.
28. Liang, H.; Li, M.; Chen, Y.; Jiang, L.; Xie, Z.; Yang, T. Establishing trusted i/o paths for sgx client systems with aurora. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 1589–1600. [CrossRef]
29. Hirano, M.; Umeda, T.; Okuda, T.; Kawai, E.; Yamaguchi, S. T-pim: Trusted password input method against data stealing malware. In Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations, Las Vegas, NV, USA, 27–29 April 2009; pp. 429–434.

30. Rutkowska, J.; Wojtczuk, R. Qubes OS architecture. *Invis. Things Lab Tech. Rep.* **2010**, *54*, 65.
31. Deshane, T.; Shepherd, Z.; Matthews, J.; Ben-Yehuda, M.; Shah, A.; Rao, B. *Quantitative Comparison of Xen and KVM*; Xen Summit: Boston, MA, USA, 2008; pp. 1–2.
32. Zhou, Z.; Yu, M.; Gligor, V.D. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 308–323.
33. Kiperberg, M.; Zaidenberg, N.J. HyperPass: Secure Password Input Platform. In Proceedings of the ICISSP, Online, 11–13 February 2021; pp. 580–587.
34. Ellis, C. The Best Free Screen Recorders 2021: Free Software to Capture All the Action. 2020. Available online: https://www.techradar.com/news/the-best-free-screen-recorder (accessed on 16 February 2023).
35. Hacks, M. Top 11 Powerful Keyloggers for Windows. 2019. Available online: https://hackernoon.com/top-10-powerful-keyloggers-for-windows-a18d3y9h (accessed on 16 February 2023).
36. Fonseca, J. VMware SVGA Device Developer Kit. 2007. Available online: https://github.com/prepare/vmware-svga (accessed on 16 February 2023).
37. Toulas, B. New Intel Chips Wo not Play Blu-ray Disks Due to SGX Deprecation. 2022. Available online: https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/ (accessed on 16 February 2023).
38. Sibai, F.N. Evaluating the performance of single and multiple core processors with PCMARK® 05 and benchmark analysis. *ACM SIGMETRICS Perform. Eval. Rev.* **2008**, *35*, 62–71. [CrossRef]
39. Lopes, F. Interception. 2015. Available online: https://github.com/oblitum/Interception/tree/v1.0.1 (accessed on 16 February 2023).