PROTECTION: Root-of-Trust for IO in Compromised Platforms

Aritra Dhar
ETH Zurich

Enis Ulqinaku
ETH Zurich

Kari Kostiainen
ETH Zurich

Srdjan Capkun
ETH Zurich

***Abstract*—Security and safety-critical remote applications such as e-voting, online banking, industrial control systems and medical devices rely upon user interaction that is typically performed through web applications. Trusted path to such remote systems is critical in the presence of an attacker that controls the computer that the user operates. Such an attacker can observe and modify any IO data without being detected by the user or the server. We investigate the security of previous research proposals and observe several drawbacks that make them vulnerable to attacks. Based on these observations we identify novel requirements for secure IO operation in the presence of a compromised host.**

As a solution, we propose PROTECTION, a system that ensures IO integrity using a trusted low-TCB device that sits between the attacker-controlled host and the IO devices. PROTECTION intercepts the display signal and user inputs from the keyboard and mouse, and overlays secure UI on top of the HDMI frames generated by the untrusted host. The guiding design principles of PROTECTION are that (i) integrity of user input and output cannot be considered separately, (ii) all user input modalities need to be protected simultaneously, and (iii) integrity protection should not rely on error prone user tasks like checking the presence of security indicators. By following these guidelines, PROTECTION achieves strong protection for IO integrity. We also propose an extension of PROTECTION for IO confidentiality and implement a plug-and-play prototype and evaluate its performance.

I. INTRODUCTION

Web-based interfaces are very prevalent to remotely configure safety-critical systems such as remote PLCs [1] or medical devices [2], and other security-sensitive applications such as online payments, e-voting, etc. The high complexity of modern operating systems, software, and hardware components has shown that computer systems largely remain vulnerable to attacks. A compromised computer threatens the integrity and the confidentiality of any interaction between the user and a remote server. It can easily alter the data exchanged between the user and the remote server, trick the user to perform unintended actions, or observe any sensitive IO data.

The recent introduction of trusted computing architectures like Intel's SGX has enabled secure computations and secure data storage on otherwise untrusted computing platforms. However, such architectures do not directly enable secure user interaction because IO operations are handled by the operating system. Additionally, the recent microarchitectural attacks have shown that execution environments inside enclaves, like the one provided by SGX, can be compromised as well.

Trusted path provides a secure channel between the user (specifically human interface device - HID) and the end-point, which is typically a trustworthy application running on the host. Trusted path ensures that user inputs reach the intended application unmodified, and all the outputs presented to the user are generated by the legitimate application. Trusted path to the local host is a well-researched area where many solutions focus on using trusted software components such as a trusted hypervisor. Zhou et al. [3] proposed a generic trusted path on $x86$ systems with a pure hypervisor-based design. SGXIO [4] employs both a hypervisor and Intel SGX. However, hypervisors are hard to deploy, have a large TCB, and are impractical in real-world scenarios as most of the existing verified hypervisors offer a minimal set of features.

Trusted external devices are another way to realize secure IO between a user and a remote server. Transaction confirmation devices [5], [6] allow the user to review her input data on a trusted device that is physically separated from the untrusted host. These approaches suffer from poor usability, security issues due to user habituation and are only limited to simple inputs. In Section II-B, we provide a more detailed discussion on the security and the usability of transaction confirmation devices. Bump in the Ether [7] and IntegriKey [8] use external embedded devices to sign input parameters. However, such solutions do not support output integrity; hence, the attacker can execute UI manipulation attacks to trick the user into providing incorrect inputs.

Fidelius [9] combines the previous ideas of Bump in the Ether and trusted overlay to protect keyboard inputs from a compromised browser using external devices and a JavaScript interpreter that runs inside an SGX enclave. Fidelius maintains overlays on display, specifically on the input text boxes to hide sensitive user inputs from the browser. We investigate the security of Fidelius and discover several issues. Fidelius imposes a high cognitive load to the users as they need to monitor continuously different security indicators (two LED lights and the status bar on the screen) to guarantee the integrity and confidentiality of the input. Furthermore, the attacker can manipulate labels of the UI elements to trick the user into providing incorrect input. The lack of mouse support, which may appear only as functional limitation, exposes Fidelius to early form submission attacks. The host can emulate a mouse click on the submit button before the user completes all fields of a form. This allows the attacker to perform an early form submission with incomplete input - a violation of input integrity. Fidelius is also vulnerable to microarchitectural attacks on SGX enclaves [10] that extract attestation keys and relay attacks [11] that relay all user data to the attacker's platform.

The drawbacks of the existing systems show that ensuring the integrity and confidentiality of the IO in the presence of an untrusted host is a non-trivial problem and requires a comprehensive solution. All of the previous trusted path solutions neither protect both input and output simultaneously, nor do they consider different modalities of input. We discuss such drawbacks in details, along with some of the relevant solutions in Section II-B.

Our solution. The shortcomings of the existing literature provide the groundwork of our system named PROTECTIOn. PROTECTIOn is built on the following observations: i) input integrity is possible only when both input and output integrity are ensured simultaneously, ii) all the input modalities are needed to be protected as they influence each other, and iii) high cognitive load results in user habituation errors. PROTECTIOn uses a trusted low-TCB auxiliary device that we call IOHUB which works as a mediator between all user IO devices and the untrusted host. Instead of implementing a separate network interface, the IOHUB uses the host as an untrusted transport - reducing attack surface.

Integrity. PROTECTIOn ensures *output integrity* by sending an encoded UI to the host that only the IOHUB can overlay on a small part of the screen. The overlay is possible as the IOHUB intercepts the display signal between the host and the monitor. The overlay generated by the IOHUB ensures that the host can neither observe nor manipulate any output information on that overlaid part of the screen; hence, it can not trick the user. IOHUB supports a subset of HTML5 UI elements that are frequently used in the majority of web applications. The IOHUB focuses user attention on the overlaid part of the screen by dimming out the rest (also known as the lightbox technique which is one of the possible ways to focus user attention) when the user moves the mouse pointer on the overlaid UI. By doing so, PROTECTIOn aids the user to be more attentive to the security-critical UI on the screen. Note that PROTECTIOn does not require any change in the user interaction for IO integrity. Only the input devices that are connected to the IOHUB can interact with the overlaid UI elements, making them completely isolated from the untrusted host. All the inputs are signed by the IOHUB and sent to the remote server

Confidentiality. PROTECTIOn provides IO confidentiality as i) all the input to the IOHUB is encrypted and signed, and ii) the overlay information sent from the remote server is encrypted and can only be decrypted by the IOHUB. However, the user needs to perform a small task such as triggering a secure attention sequence (SAS), or looking for a secret image, security indicator etc. to distinguish the trusted overlay.

Deployment. IOHUB is a fully plug-and-play device that is compatible with any host system regardless of their architecture or OS and does not require the user to install any software on the host. Note that our realization of PROTECTIOn uses an external device. However, the current system architecture can be modified, e.g., IOHUB can be integrated into the graphics processor.

Our contributions. We now summarize our contributions:

(i) Identification of IO security requirements: We identify new requirements for trusted path based on the drawbacks of the existing literature: i) unless both output and input integrity are secured simultaneously, it is impossible to achieve any one of the two, and ii) without protecting the integrity of all the modalities of inputs, none could be achieved (Section II-B).

(ii) System for IO integrity: We describe the design of PROTECTIOn, a system that provides a remote trusted path from the server to the user, in an attacker-controlled environment. The design of PROTECTIOn leverages a small, low-TCB auxiliary device that acts as a *root-of-trust* for the IO. PROTECTIOn ensures the integrity of the UI, specifically the integrity of mouse pointer and keyboard input. PROTECTIOn is further designed to avoid user habituation (Sections III and IV).

(iii) System for IO confidentiality: We also describe an extension of PROTECTIOn that provides IO confidentiality, where user needs to execute an operation like SAS to identify the trusted overlay on the display (Section V).

(iv) Implementation and evaluation: We also implement a prototype of PROTECTIOn and evaluate its performance (Sections VII and VIII).

II. PROBLEM STATEMENT

In this section, we motivate our work in the context of ensuring the integrity and confidentiality of IO data between the user and the remote servers. We also analyze existing research works that tackle the relevant problem. We explain how these works lack a proper solution and report the observations we derive from them. Lastly, we present the required security properties of PROTECTIOn that we obtain from the observations.

A. Motivation: Secure IO with Remote Safety-critical System

A user communicates with a remote server through a *host* system that is typically a standard PC (specifically $x86$ architecture), which gives the host access to the raw IO data that is exchanged between the user and the remote server. The host consists of large and complex system software such as the operating system, device drivers, applications such as browser, and a diverse set of hardware components that expose the host to a large attack surface. An adversary that controls the user's host can alter user intentions, i.e., it can perform arbitrary actions on behalf of the user, modify the input parameters, or show wrong information to the user. Such an adversary is very powerful and difficult to be detected or prevented by a remote server. Hence, existing defense standards for web UI are ineffective as the browser is untrusted also. The consequences of such attacks might be severe when applications that control remote safety-critical systems are targeted. The attacker can pass the wrong input to a remote safety-critical system such as a medical device, power plant, etc., or leak sensitive information such as credentials for e-banking, candidate preference in the e-voting, etc.

B. Analysis of Existing and Strawman Solutions

There are two broad categories of existing solutions that address the problem of trusted paths for IO devices in the presence of a compromised host as illustrated in Figure 1: **A.** Solutions where unprotected user interaction first happens and then a trusted component (transaction confirmation device) is used to ensure input integrity, and **B.** Solutions where a trusted component captures the user's input/output and then securely mediates them to the destination. The trusted component can be a hypervisor, or an external hardware, etc.

A. Transaction confirmation devices. Filyanov et. al [5] proposed transaction confirmation device that requires the

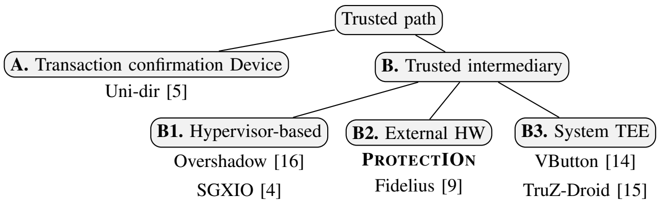


Fig. 1: **Existing trusted path solutions** by their approach.

user to use a separate device to confirm the input parameters. Systems such as ZTIC [6] use an external device with display and smartcard attachment to ensure the integrity of the user inputs. Android OS also provides a similar mechanism to confirm protected transactions [12]. However, these approaches suffer from three significant drawbacks: i) the risk of *user habituation* – users confirming transactions without looking to the actual data [13], ii) *usability* – interacting with a small device can be cumbersome, and iii) only *simple UI* can be supported – transaction confirmation is not suitable for complex interaction, rather than simple text-based inputs.

B1. Trusted hypervisor-based solutions. Trusted hypervisors and secure micro-kernels are also alternatives to achieve Trusted path. Zhou et al. [3] proposed a generic trusted path on $x86$ systems in pure hypervisor-based design. SGXIO [4] combines a TEE and a hypervisor to mitigate the shortcomings of TEEs like SGX (e.g., OS controls the IO operations). Nevertheless, solutions based on hypervisors require a large TCB. Formally verified hypervisors offer limited functionalities, therefore making them impractical for average users. One can also argue that a hypervisor that provides a rich set of functionalities has a code size comparable to an actual OS. Also, systems employing TEEs such as Intel SGX open up new attack surfaces that can be exploited by microarchitectural attacks [10].

B2. External hardware-based solutions. Several existing works propose a trusted path that utilizes an external trusted device. IntegriKey [8] uses a trusted external device that contains a small program which signs all user inputs and sends the signed input to the remote server. The device works as a second factor for input integrity as the remote server verifies if the signed input matches with the input that is sent by the browser running on the untrusted host. However, as the external device is completely oblivious to the display information that the untrusted host renders, not only IntegriKey but also similar systems that do not consider output integrity are vulnerable to UI manipulation attacks. For example, assume that the user's intended input to a textbox is 100. She types the correct value, but the host maliciously renders 10 on the screen by not showing the last zero. Thinking that she might have mistyped, the user types another 0 that makes the recorded input from the user 1000. This attack violates input integrity as the host can now submit 1000 to the remote server as a valid input, although it does not represent the user's intention.

$\rightarrow$ *Observation 1:* The lack of output integrity – *the render of user inputs on the screen* – compromises input integrity.

Fidelius [9] addresses the problem with output integrity by rendering overlays using an external trusted device. Fidelius uses the trusted external device and Intel SGX to create a secure channel between the user IO devices and a remote server. The device intercepts user keystrokes and does not deliver any event to the untrusted host when the user types to secured text fields. Additionally, Fidelius renders an overlay with the user inputs on the screen, which is inaccessible by the host. This way, the untrusted host does not have access to raw inputs while the user sees them rendered on the screen as usual. A small, trusted bar on display is also overlaid by the device that shows the remote server's identity and the text field that is currently selected. However, we observe a number of security and functional issues in Fidelius that we explain in the following.

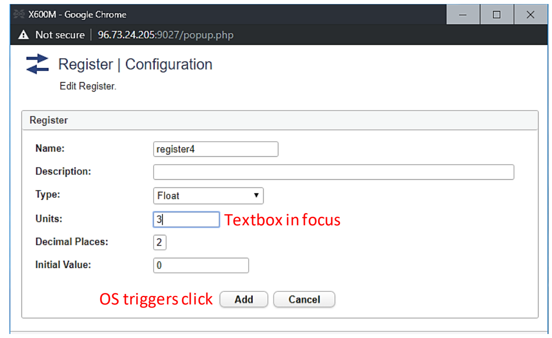


Fig. 2: **Early form submission attack** is possible on Fidelius [9]. The user selects and edits the field `Units` while the OS triggers `add` button, causing misconfiguration of a remote safety-critical PLC (Control by Web X-600M [1]).

The overlay contains only the render of the user inputs into text fields, but the rest of the screen is rendered by the untrusted host. This allows an attacker to modify the instructions on the UI, such as changing the unit of the input (typically described in the label of a text field) that could result in an incorrect input. This problem could be mitigated if the trusted bar includes the legitimate labels of the text fields also, although it would significantly increase the cognitive load to users.

Fidelius already introduces a high cognitive load to users as they need to monitor multiple security indicators simultaneously before filling up one text field. Previous research works [13], [17], [18] have shown that systems that require users to observe multiple security indicators do not guarantee security in practice. Also, in specific scenarios, even the training to properly explain these indicators to users could be a significant drawback for a real deployment.

$\rightarrow$ *Observation 2*: If the *protected output* is provided out-of-context, users are more likely not to verify it. Therefore input integrity can be violated.

Fidelius does not consider the integrity of the mouse pointer and its interaction with UI elements which broadens the attack surface. The lack of mouse support may appear to be a functional limitation, but it has non trivial security issues. The OS can arbitrarily trigger a mouse click on the submit button of a form while the user is typing and therefore send incomplete data to the server - early form submission attack. This attack could cause the misconfiguration of a remote system, as illustrated in Figure 2. Early form submission may appear to be similar to clickJacking attack, but the fundamental difference between them is that in clickjacking the browser and OS are considered to be trusted. An untrusted OS can simply issue mouse clicks.

Moreover, Fidelius is also vulnerable to clickjacking attacks where the attacker can spawn a fake mouse pointer and trick the user into following it while the real mouse pointer is on a sensitive text field protected by the system. This allows the attacker to fool the user into providing (possibly incorrect) input, while the user thinks that she is interacting with a non-sensitive text field. To prevent such attacks, the user has to look at the security indicators continuously even when she is not doing any security-sensitive task, which is a very strong assumption. Thus, not supporting the mouse causes the integrity violation of the keyboard input also.

$\rightarrow$ *Observation 3:* If not *all the modalities of inputs* are secured simultaneously, none of them can be fully secured.

Finally, the design of Fidelius [9] is strictly limited to text-based fields only. As Fidelius does not provide output integrity of the forms, it cannot provide confidentiality to other UI elements such as radio buttons, drop-down menus, sliders, etc. Microarchitectural attacks on Intel SGX increase significantly the attack surface of the system also [10].

B3. System TEE-based solutions. VButton [14] uses ARM TrustZone (TZ) to securely render UI buttons and receive user input from them. This is possible on mobile devices, because the TZ architecture support flags on the system bus that indicate whether an IO device like touchscreen communicates with a trusted TZ application or the untrusted OS. Such solutions are infeasible for us because i) secure communication between IO peripherals and TEE applications (like SGX enclaves) is not supported in the x86 architecture – a similar system in x86 would require changes to the system architecture, TEE architecture and IO devices, ii) such solutions require TEE-aware applications and do not work with current browsers. Our goal is to design a solution that can be deployed on current the x86 architecture and used with existing popular browsers.

Strawman solution: Capturing screenshot. This strawman solution uses a trusted device that takes a screenshot when the user executes an action, e.g., mouse click to submit a form. The device then signs the snapshot and transmits it to the server along with the signed input. The remote server verifies the signature and then uses image/text analysis to extract the information from the UI elements such as labels on buttons or markers of a slider, etc. Therefore, the server would detect if the host has manipulated UI elements when presented to the user.

This method is vulnerable to attacks because it does not capture the spatiotemporal user context. This implies that the attacker may show some spacial information on the screen to influence the user that may not be captured by the snapshot. Furthermore, taking a full-screen snapshot could also reveal private information of the user from other applications. Similarly, taking a snapshot does not guarantee that a specific UI has been presented on the screen as the attacker may render the legitimate UI shortly before the device captures the snapshot. One way to mitigate this problem is to capture a video of user interaction. But such a method requires the host to send large amounts of data to the server, while the server should support video processing for different browsers which is both time and CPU intensive. Lastly, adversarial machine learning techniques [19], [20] make the image/text recognition techniques insecure against advanced adversaries.

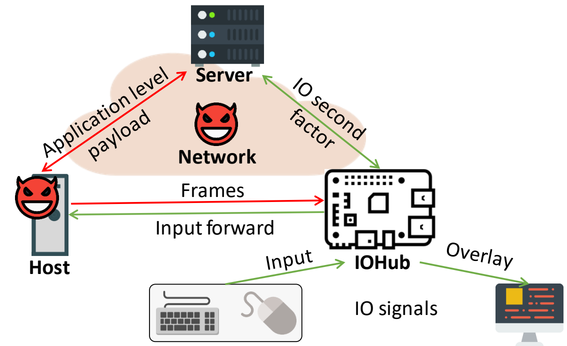


Fig. 3: **High-level approach overview of our solution.** The IOHub connects the trusted IO devices and the attacker-controlled host.

C. Requirements of Security and Functional Properties

The lack of security properties and features in the existing solutions provides the necessary security and functional requirements for a trusted path that provides IO integrity and confidentiality and is usable. We can now summarize the observations that we derived from the literature and the strawman solution (refer to Section II-B) as following:

R1. Inter-dependency between input and output. The first and second observations from the existing solutions show that the output and input security depend on each other, and they should be considered together. Otherwise, the attacker can manipulate the output to influence the user input.

R2. Inter-dependency between all input modalities. Existing web interfaces allow users to complete forms by using different modalities for the user input, namely the keyboard, the mouse, and the touchpad. The third observation shows that a secure system should protect simultaneously all user input modalities to achieve input integrity (against early-form submission and clickjacking).

R3a. No cognitive load for IO integrity. A system that protects IO operations should introduce minimal or no cognitive load to its users for input integrity. The system should guarantee the output integrity of the legitimate information necessary to complete a form and avoid asking the user to interact with an external device or monitor security indicators out-of-context.

R3b. User attention for IO confidentiality. Preserving the confidentiality of user inputs against a compromised host is a challenging task because the host can trick the user to reveal her inputs when the system is not active. Therefore, requiring users to perform a small action, e.g., press a key, before entering confidential inputs is a valid trade-off between usability and security.

R4. Small trust assumptions and deployability. Our goal is to provide the rich set of IO and security features with minimal trust assumptions that do not rely on a trusted OS, specialized hypervisor, or TEEs such as Intel SGX. Preferably, the solution should be easy to set up for users, i.e., plug-and-play, and integrate well with the existing infrastructure.

III. System Overview & Main Techniques

In this section, we present an overview of our solution: ProtectIOn. On the high-level, ProtectIOn uses the concept of the *bump in the wire* (such as bump in the ether [7]) to provide integrity and confidentiality to the user IOs between the IO devices and the remote server. ProtectIOn achieves this by utilizing a trusted embedded device as a mediator between all the IO devices and the untrusted host. Hence, our approach falls into the category **B2** (external HW) in Figure 1. We call this trusted intermediary IOHub for the rest of this paper.

A. System and Attacker Model

We consider a typical scenario where the user wants to interact with a trusted remote web server via an attacker-controlled host. The model is depicted in Figure 3 that shows the untrusted host, the remote server, and the user IO devices. We only assume that the monitor, keyboard, mouse (in a word all the IO devices that we need to protect from the malicious host) and the IOHub are trusted. The IOHub works as a mediator between all the IO devices and the host. Note that the IOHub has no network capability to communicate with the server directly, rather it relies on the host and uses it as an untrusted transport. We also assume that the IOHub comes with preloaded certificates and keys that allow the IOHub to verify the signatures signed by the server and sign data such as the user input.

There are many possible ways to deploy ProtectIOn. One way is to assume that the IOHub manufacturer issues a certificate for each of the deployed IOHubs . The IOHub maintains a whitelist for the remote servers along with their public certificates. This allows the IOHub to verify messages signed by those remote servers. Another assumption could be that the IOHub is issued by a service provider who also runs the remote server.

Attacker model and capabilities. Our attacker model assumes that the host (OS, installed applications, and hardware) and the network are attacker-controlled. The attacker can intercept, and arbitrarily manipulate (such as create, drop, or modify) the user IO data between the user and the remote server. Furthermore, we assume that the attacker can not break the physical security of the IOHub (more discussion in Section VI-C).

B. High-level Description of the System

ProtectIOn is build upon the required security and functional properties that are described in Section II-C. IOHub is active only when the user visits sensitive web applications that require ProtectIOn security. Initially, the remote server signs and delivers the sensitive UI elements to the host in a format that is understandable by IOHub. Next, the host transfers the sensitive UI to IOHub, and the IOHub verifies the signature to prevent manipulations by the host. As seen in a running example depicted in Figure 4, the IOHub then renders the UI with sensitive elements into an overlay on top of the HDMI frame received from the host. Note that the host cannot access or modify the overlay generated by the IOHub. Also, the overlay covers only a part of the screen, allowing the other feature-rich content on the webpage to run unmodified. Therefore, this ensures that sensitive UI elements are presented to the user as expected by the remote server – *output integrity*. For the overlay, we use QR-codes to transfer data from the host to the device because we avoid using extra software/hardware for a separate channel, and it is easy to visualize.

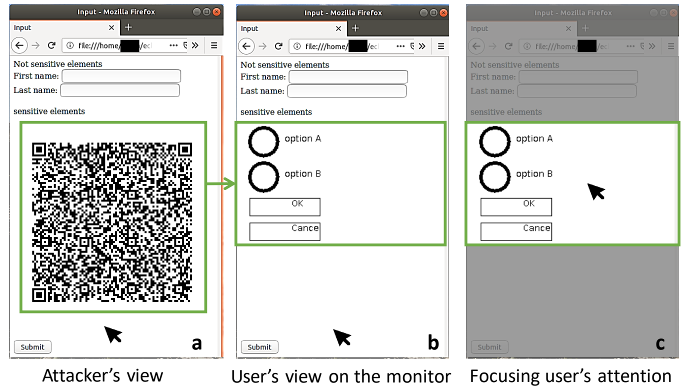


Fig. 4: **ProtectIOn's high-level approach** shows that the IOHub generates UI overlay to protect IO integrity and confidentiality. a) The attacker only sees the non-protected UI elements, and the protected form is encrypted and encoded (in our case, the IOHub could decode a QR code and decrypt). b) shows the IOHub generated form overlay that is hidden from the host. The protected part of the screen provides integrity and confidentiality of all user IO. c) shows that the IOHub dims out (lightbox) the rest of the screen when the user moves her mouse pointer over the protected region to focus user attention.

When the user interacts (types or moves the pointer) with the overlay, IOHub does not forward any event from the keyboard or the mouse to the host. The interaction is maintained solely by IOHub, which renders on-screen user inputs and therefore offers a user experience that is identical to a typical one as if the IOHub is not present. The user click on the *submit* button triggers the submission procedure, which consists of the IOHub signing the user inputs and sending to the server. Note that the text fields of the form and the *submit* button are inside the overlay which is inaccessible by the host, hence the attacker cannot execute the early form submission or clickjacking attacks. Finally, the server verifies the signature of IOHub to guarantee that the host has not altered the data. Therefore, the IOHub ensures *input integrity* for all *modalities* of input.

For integrity protection, ProtectIOn uses well-known user attention focusing mechanisms. Unlike systems like Fidelius, these mechanisms do not introduce any cognitive load to the users as ProtectIOn does not rely on multiple security indicators. Mechanisms such as lightbox aid the user to distinguish the IOHub overlay on the screen from the rest. Thus, the untrusted host cannot trick the user into following malicious instructions when the user interacts with sensitive UI elements. Also, the host cannot observe sensitive data on the overlay because it does not have access to it. In the case where confidentiality is required, the user manually triggers SAS, such as the lightbox by pressing specific keys.

IV. ProtectIOn for IO Integrity

In this section, we provide the technical details of ProtectIOn integrity protection for IO devices.

A. IOHub *Overlay of UI Elements*

As we explained in the previous sections, both output and input integrity are necessary to be protected to achieve any of them. ProtectIOn ensures output integrity by isolating a part of the display that cannot be observed or modified by the untrusted host. IOHub intercepts the HDMI frame from the

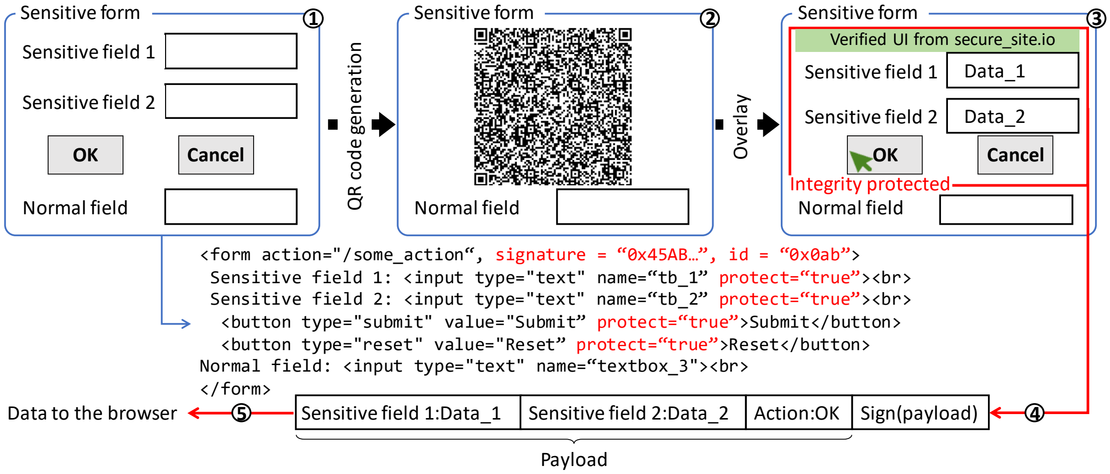


Fig. 5: **Transformation of UI elements: HTML → encoded specification → IOHUB generated UI overlay.** ① The actual webpage and the corresponding HTML source shows the UI elements that requires integrity protection. ② These UI elements are transformed into an encoded UI specification (our PROTECTION prototype uses QR code that encodes a UI specification, e.g., Specification 1) by the PROTECTION JS. The QR code. ③ AThe QR code decoded and overlaid on the HDMI stream by the IOHUB. ④ Upon the user's action on the overlaid UI elements, the device signs all the input data. ⑤ The IOHUB sends these signed input data them to the remote server. Note that the intermediate QR code transformation (②) is not visible by the user as it is decoded instantaneously by the device.

Specification 1: **Protected UI specification language.** The UI specification shows the JSON formatted UI specification that is generated from the HTML source provided in the UI illustrated in Figure 5.

```
{"formId": "form1", "formName": "form1",
 "domain": "secure_site.io",
 "size": "400*400", "SAS": "ctrl+d:5",
 "ui": [{"id":"textbox_1", "type":"textbox",
    "label":"Sensitive field 1",
   "text":"secret data 1",
   "RE":(A-Z)*.(A-Z)*},
   {"id":"textbox_2", "type":"textbox",
   "label":"Sensitive field2 ",
   "text":"secret data 2"},
   {"id":"b1", "type":"button",
   "label":"OK", "trigger":"true"},
   {"id":"b2", "type":"button",
   "label":"Cancel", "trigger":"false"}],
 "signature": "0x45AB...", "id": "0x0ab.."}
```

host and injects a render of the sensitive UI on the screen. The overlay provides output integrity because it restrains the attacker from drawing on top of it to trick the user into providing incorrect inputs.

To minimize the TCB, the IOHUB does not run a browser, i.e., it can not interpret or render HTML, JavaScript, etc. Instead, the IOHUB comes with a small interpreter routine that is similar to browser renders engines in functionality, but drastically smaller in size because it only renders a limited number of HTML5 UI elements according to their position, dimension, and label. The interpreter routine reads a given specification and renders the respective UI. The specification is a simple *JSON* file that defines how the content of the overlay should be rendered, e.g., number of elements, order, types, and labels.

The process of rendering the overlay on the screen has two phases: (i) convert the existing sensitive form to specification, and (ii) specification to overlay.

(i) Secure form → Specification. The W3C UI security policy [21] recommends developers to annotate the security-critical UI elements of a page to protect them against malicious JS running on the browser. We use a similar technique by asking developers to manually annotate the sensitive elements in the HTML code (as `protect="true"` attribute). Then For every request, the PROTECTION server-side component parses the HTML source, adds a random identifier (`id`) to the `form` element, signs it, add the `signature` to the `form` and then delivers it to the user's browser. The `id` serves as session identification to prevent the attacker from re-submitting an old input data from the user. On the client side, PROTECTION JS parses the tagged HTML source and produces a specification that could be interpreted by the IOHUB. An example of a specification is presented in Specification 1. In our implementation, the PROTECTION JS encodes the specification in a QR code. Figure 5 shows the transformation between the step ① and ②. The step ② is processed by IOHUB in the next phase and is not visible to the user.

(ii) Specification → Overlay. IOHUB performs the next phase, which starts with the detection of the encoded specification (QR-code) in the HDMI frames. Then the IOHUB validates the signature, renders the overlay according to the specifications and presents it to the user. The IOHUB overlay is depicted in ③ in Figure 5, which is the final UI shown to the user. Note that the user does not see the QR code as it gets decoded and overlaid by the IOHUB on the fly.

IOHUB uses the specification to determine the particular UI element that the user interacts with. When the user clicks on a text field, IOHUB allows the user to type input to it. UI elements in the overlay take inputs only from input devices (mouse and keyboard). Therefore a malicious host cannot inject or modify any input of the user.

B. Focusing User Attention

In the previous section, we explain how PROTECTION provides output integrity for the overlay generated by the IOHUB. However, the attacker can show fake information to the user on the untrusted part of the display space that may potentially influence her inputs. An advanced adversary could craft malicious directions and present to the user as part of the overlay.

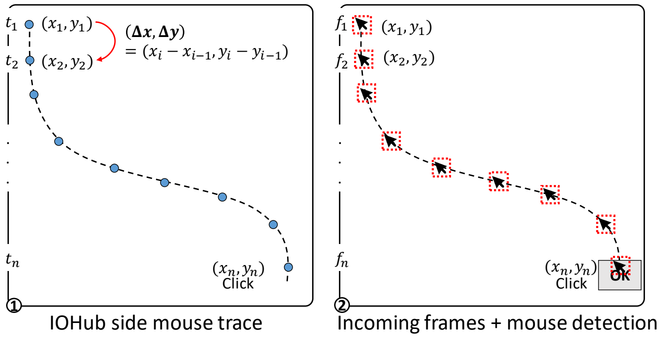


Fig. 6: **Pointer tracking.** ① The IOHUB captures the raw mouse events $(\Delta x, \Delta y)$ from the mouse that is attached to the IOHUB. ② The IOHUB captures the frames from the HDMI channel and checks into the designated pixel position $(x_i+\Delta x, y_i+\Delta y)$ if there exists a pointer. $t_1, t_2, \ldots t_n$ are the time instances when the IOHUB receives the mouse data. $f_1, f_2, \ldots f_n$ are the corresponding HDMI frames that the IOHUB intercepts.

To mitigate these attacks, we employ techniques that are proposed against similar threats in the context of browser-based security. The goal of these techniques is to focus user attention to the sensitive UI elements she is interacting with. Huang et al. [22] proposes two main techniques that are shown to be effective and can easily be adopted by the IOHUB. The first technique is called Lightbox, and it dims out non-overlaid part of the screen, which is generated by the untrusted host. The second technique consists of freezing display frames from the host when the user enters into the overlaid UI. This way, a malicious host cannot grab the user's attention by showing an animation or exploiting other tricks. Lightbox offers more security guarantees because it blocks the untrusted screen completely, but is more intrusive to the user. While freezing is less intrusive but does not remove potential malicious information from the screen. Lightbox mitigates the attacks presented above. The paper shows that the lightbox and freezing are effective in 98% and 97% of the time (baseline: 69% effectiveness when no protection is provided) respectively, making them suitable candidates for PROTECTION. For more details of the user study, refer to Table 2 in [22]. We assume that similar result should be expected in PROTECTION due to the similarity of the application space (web applications). IOHUB uses Lightbox as the default technique, but depending on the specific form, the developers can select the appropriate technique.

Automated activation. The technique to focus user attention (dimming out or freezing the non-overlaid part of the screen) is triggered automatically in specific situations: The user moves the mouse pointer over the overlaid UI, or the user starts typing into a sensitive UI element. The advantage of the automated trigger is that the user does not need to remember to activate the mechanism. Hence the system is resilient from user habituation and does not require the user to actively monitor security indicators or perform specific actions. Note that the automated activation provide security to user IO data *only* when integrity of the data is considered.

C. Continuous Tracking of Mouse Pointer in the HDMI Frame

The triggering of the focusing mechanism poses a challenging task to PROTECTION because the IOHUB does not know the exact position of the mouse pointer. We cannot rely on the compromised host to communicate the pointer position reliably to IOHUB. Furthermore, the host's pointer is not visible when the user interacts with the overlay rendered by the IOHUB as the IOHUB always draws on top of the HDMI frames of the host.

IOHUB could employ image analysis over the frame received from the host to learn the pointer position. However, we avoid this method because image analysis is time-consuming and vulnerable to adversarial images. In our approach, the IOHUB intercepts mouse events and HDMI frames, so it can track the pointer based on mouse data and correlate it with the actual position in the HDMI frame (using shape detection in a small rectangle). Then, the IOHUB overlays a mouse pointer that is prominent and easy to follow by the user.

A malicious host can still show a fake pointer to trick the user into following it, but when the focusing mechanism is active (the user interacting with sensitive elements), only the pointer overlaid by IOHUB is visible. This way, the pointer tracking and the pointer overlay address three major challenges: i) both the IOHUB and the user have the same sense of the pointer position, ii) IOHUB knows precisely when to trigger the focusing mechanism, and iii) the user can interact with the overlaid UI seamlessly.

*1) **Calibration:*** When the user connects the IOHUB for the first time after booting up, the IOHUB performs an automated calibration to find the pointer. The IOHUB simulates the mouse and pushes the pointer to the top-right corner of the screen. Then the IOHUB searches the pointer at this position in the HDMI frames and starts tracking the pointer afterward. Note, that at any point, if the IOHUB loses track of the mouse pointer, the *calibration* process is repeated the first moment the user visits a website that employs PROTECTION.

*2) **Pointer detection:*** The IOHUB ensures pointer integrity by tracking the mouse movements using the raw data from the mouse and the HDMI frame. Figure 6 illustrates the high-level idea:

① Shows raw mouse data that notify the displacement events $(\Delta x, \Delta y)$ over x and y axis which are fired over time series $t_1, \ldots, t_n$. Note that the initial pointer position is known to the IOHUB from calibration phase where $(x_0, y_0) = (0,0)$.

② Shows the HDMI frames $f_1, \ldots f_n$ where the IOHUB expects the mouse pointer to be found. For efficiency, the IOHUB only scans a small portion of the HDMI frames (200×200 square pixels) that is enough to cover a mouse pointer. Since the operating system can treat mouse movements slightly different according to their algorithm, this step serves to adjust the position difference.

*3) **Overlay of the mouse pointer:*** The IOHUB draws a mouse pointer overlay on top of the actual mouse pointer. The host mouse pointer is neither visible on top of the overlay not it can interact with the IOHUB's overlay. The overlaid mouse pointer is visible on top of the overlay, and it offers the same user experience as the host-rendered mouse pointer.

*4) **Coping with the disappearing pointer:*** Many OS offer a feature where the mouse pointer disappears from the screen when the user types in a text editor/browser. When the user moves her mouse, the cursor appears again at the same

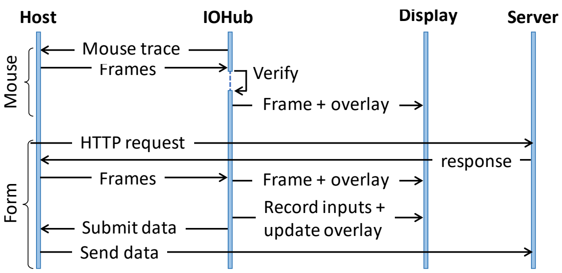


Fig. 7: **Flow of the ProtectIOn main protocol.** The figure shows the sequence of events for two example scenarios: mouse movement and filling up a web-form.

position where it disappeared in the first place. From the IOHub's perspective, it is hard to distinguish between this case and the attacker deliberately removing the mouse pointer from the screen. To handle this case, the IOHub listens to all the keyboard inputs – the keyboard is also connected to the IOHub. Therefore, when the IOHub gets a keystroke event, it expects the cursor to disappear from the screen. Then, IOHub continues tracking the pointer from the moment that the mouse sends events - this way, the IOHub ensures the consistency of the pointer position.

*5) **Handling different mouse cursors:*** The IOHub is preloaded with template images of the mouse pointer for detection. For our ProtectIOn prototype implementation, we use the default cursors provided by the Ubuntu OS. This allows the IOHub to identify the cursor when it changes on the screen, e.g., from pointer to a hand when the user hovers the pointer over a link on the browser.

*6) **Handling mouse acceleration:*** The IOHub uses the default mouse acceleration parameters of `libinput` to cope with the pointer acceleration. As the IOHub emulate itself as a keyboard, at the time of initialization, the IOHub sends a command to the host to set the default acceleration. In case, the host changes the mouse acceleration; the IOHub will fail to detect the mouse in the HDMI stream. We consider this case as a denial of service.

D. Protected User Interaction

When the user finishes providing her input via input devices (mouse and keyboard), the IOHub sends these values (with signature to ensure integrity) to the remote server. Sending these signed input values to the server requires an upstream channel from the IOHub to the server.

Upstream channel. The data from the IOHub to the remote server is transmitted using the ProtectIOn JavaScript snippet as a helper. The IOHub emulates itself as a composite human interface device (HID) when it is connected to the host. The IOHub emulates keystrokes that transmit encoded data to the ProtectIOn JavaScript snippet, which then forwards them to the remote server.

Sending input data. Figure 7 depicts the user interactions in a sequence diagram.. The user input transmission procedure is illustrated in Figure 5. This has two phases: *record* and *transmit* as described in the following:

(**i**) **Record.** After the UI elements are correctly overlaid on the screen, the users can interact with these UI elements. The user interaction with the overlaid UI element is no different than a standard UI. The UI specification encodes the behavior of all generated UI elements, making the IOHub aware of the semantics of the UI objects. E.g., when a user selects a text box and types on with her keyboard, the IOHub intercepts all keystrokes and renders the characters on the overlay. When user enters input data in the rendered overlay UI elements (such as textbox, button, slider, radio button, etc.), the IOHub records that in a (key, value) pair where the key is the identifier of the UI element (`id` in Specification 1) and the value is the user provided value. The `type` of the UI elements determines what information to record. For example, the IOHub records all keystrokes when a textbox is selected, the value corresponding to the position of the slider is recorded when the user interacts with a slider, etc. One example of the recording of the input data corresponding to the UI illustrated in Figure 5 and Specification 1 is:

$$Record = (tb\_1, Data\_1); (tb\_2, Data\_2)$$

(**ii**) **Transmit.** In the transmit phase, the IOHub waits for the user to select a UI element which has a `trigger` capability, e.g., a submit button on a web-form. A trigger element can change the state of the overlaid form, e.g., submit the data of the form to the remote server or reset it. More details are provided in the implementation of ProtectIOn in Section VII-A1. When the user clicks the `OK` button, the device signs $Record$ with its embedded private key. One such signed packet is also illustrated in Figure 5. The IOHub sends the signed packet to the remote server using the upstream channel.

Upon receiving the signed input data from the IOHub, the remote accepts the input if the signature verification is successful. Note, if an input field is annotated as `protect="true"`, the server does not accept any input without the IOHub signature. This prevents the attacker-controlled host to submit data.

Changing browser tabs or browsers. The IOHub supports multiple browsing tabs across multiple browsers. The UI specification contains `formId` and `domain` that works as the unique identifier for a specific form served from a specific web server. The IOHub can maintain multiple parallel TLS connection to web servers. Depending on the observed `formId` and `domain` (refer to Specification 1), the device retrieves the data that is entered by the user. This way even if the user switches tabs, the IOHub can still allow editing the forms across tabs.

Input validation. Input validation, i.e., checking the input against a recommended input policy (e.g., regular expression) is one of the most widely used JavaScript functionalities and it is a critical part of input integrity. The remote server sends the regular expression in the UI specification (`RE` in Specification 1) that the IOHub uses to validate the user input.

Fallback for legacy clients. ProtectIOn is backward-compatible with the clients who do not use the IOHub. This is achieved by the remote server by showing a QR code briefly on the screen when the user visits the ProtectIOn-enabled webpage. The IOHub intercepted the QR code and sends a signal to the server about its presence. In the absence of the IOHub, the remote server does not send the ProtectIOn JS to the host that acts as a communication channel between the IOHub and the remote server. Note, that the fallback

Specification 2: **HTML page from the remote server that contains the encrypted UI specification for IO confidentiality.**

```
<form action="/some_action">
  Text box 1:<br>
  <input type="text" name="text_box_1">
  <br> text box 2:<br>
  <input type="text" name="text_box_2">
  <encrypted_qr><!-encrypted UI specification->
  0x4a5c4... </encrypted_qr>
  <script> [JS outputs QR code that encodes
  encrypted specification] </script>
</form>
```

mechanism is application-specific and the service provider could decide if the fallback is detrimental to security.

V. PROTECTION FOR IO CONFIDENTIALITY

In the previous sections, we describe how the PROTECTION JavaScript and the IOHUB together ensure the integrity of the IO. We now augment the design of PROTECTION to achieve IO confidentiality alongside the IO integrity. One of the major components for achieving IO confidentiality is to establish a secure channel (i.e., a TLS channel) between the remote server and the IOHUB. TLS ensures that the untrusted host does not read or modify any data exchanged between the user and the remote server.

A. IO Operations

Establishing TLS. The IOHUB and the server create TLS using the public certificates. The TLS uses the emulated keystroke streams and HDMI as the upstream and downstream channels respectively as described in Section IV. Implementation details are provided in Section VII-A5.

Output confidentiality. Output confidentiality ensures that information sent from the remote server and the visual render of the user's input is hidden from the host. To enable output confidentiality, the UI overlay mechanism that is described in Section IV-A is modified slightly. The difference is that the specification is not generated in the client side, but rather in the server. A small server-side module that is very similar to PROTECTION JS transforms the UI elements to the UI specification (one example is provided in Specification 1) and encrypts it with the TLS session key. The encrypted specification is delivered to the client browser inside the <encrypted_qr> tag in the HTML file which is then encoded (as a QR-code) by the PROTECTION JS. The IOHUB decodes the QR code from the intercepted HDMI frames, decrypts the specification and renders the overlay accordingly. One example is provided in the HTML Snippet 2 with the corresponding UI illustrated in Figure 8. This feature of PROTECTION allows the remote server to send securely private information to the user in the presence of a compromised host, e.g., bank account statements, or any other confidential message.

Input Confidentiality. When the user enters her mouse pointer into the overlaid UI area, the IOHUB stops transmitting any mouse or keyboard event to the host, making it completely oblivious of any mouse movement or keystroke during that time. However, the user can still see her inputs on the screen as the IOHUB renders the plaintext character on the overlaid UI elements, therefore making them visible only to the user.

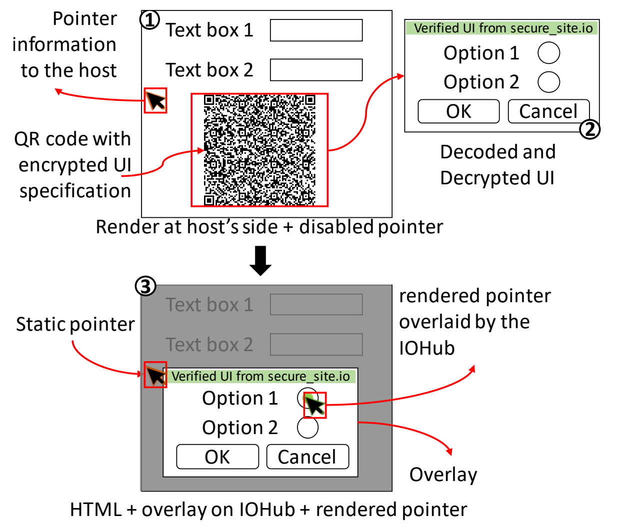


Fig. 8: **PROTECTION IO confidentiality.** The figure shows ① the browser render of the webpage in Specification 2 where the PROTECTION JavaScript produce the encrypted QR code. ② shows the UI overlay that is decrypted and decoded by the IOHUB. ③ shows the user's view when the IOHUB overlays the UI on the HDMI frame, and the user starts to interact with the UI.

Likewise, when the user selects a UI element, for example, a radio button that is shown in Figure 8, the IOHUB stores the selected value in the recorded data. On form submission, IOHUB encrypts the recorded data with the TLS key and sends them to the remote server.

B. Focusing User Attention

The IO confidentiality could be viewed as a similar problem to *phishing* where providing input to an attacker-generated UI (or a phishing webpage) leaks sensitive information. Similar to the phishing protection mechanisms, IO confidentiality requires the user for additional attention/operations. Secure Attention Sequence (SAS) is a sequence of trustworthy actions (such as keystrokes `Ctrl+Alt+Del` in Windows) executed by the user. SAS prevents an untrusted system from triggering an event that is otherwise sensitive to the user. Note that SAS is a well-researched topic in the context of UI/UX design. PROTECTION adapts an off-the-shelf SAS mechanism that provides a visual aid for the user to distinguish overlaid UI and the mouse pointer location. SAS is crucial for IO confidentiality as the untrusted host can trick the user into inputting her sensitive information on a forged form. Hence, the user needs to remember the SAS to distinguish IOHUB generated UIs from host generated UIs. Note that the automated activation is insufficient as at any given time, the host can maliciously emulate the automated activation to trick the user into providing sensitive information to an illegitimate UI.

Note that, SAS is one of a ways to inform the user securely about the trusted overlay on the screen generated by the IOHUB. Evaluation of the effectiveness of SAS over other attention focusing mechanisms is out-of-scope of this paper. Hence, PROTECTION uses SAS as an example attention focusing mechanism for confidentiality. In principle, PROTECTION could be integrated with other proposed approaches such as security indicators, or secret images [23], [24].

SAS policy. The remote server can set configurable SAS policy per overlaid UI (i.e., QR code). The SAS policy is defined in the `SAS` attribute in the example specification provided in Specification 1. By default, the overlaid UI is locked from the user and requires a key press from the user to unlock the sensitive UI. This information is overlaid on the UI to remind the user to execute it. One example policy could be `Ctrl+d:5`, which denotes that the user needs to press key '`Ctrl+d`' to unlock the UI overlay. Pressing this key also triggers the IOHUB to black out the HDMI frames except for the UI overlay and the mouse pointer overlay for a specified time (here for 5 seconds).

VI. SECURITY ANALYSIS

A. Integrity

Modifying IO operations. As only the IOHUB can interact with the overlaid UI, the attacker can not manipulate the IO operations with the overlaid UI. Moreover, the attacker cannot submit arbitrary data to the remote server because the latter accepts only inputs signed by the IOHUB.

Early form submission. This attack is not possible as the input devices (both mouse and keyboard) are connected to the IOHUB and only the IOHUB can interact with the overlaid UI. This makes it impossible for the attacker to emulate a click on the overlaid part of the screen.

Attack on the mouse pointer tracking and overlay. The attacker may try to defeat the PROTECTION pointer tracking and overlay mechanism described in Section IV-C by introducing a malicious pointer that is visually more appealing to the user. Note that the IOHUB overlaid mouse pointer is prominent and hard to miss. One can visualize it as an arms race between the attacker and the IOHUB to grab the user attention. But, we argue that this is a suboptimal strategy for the attacker as both of the pointers will be visible on the screen that cause suspicion to the user. Also, when the real mouse pointer enters the overlaid area, the untrusted part, including the malicious mouse pointer, will be hidden by the focusing mechanism. Hence, we can conclude that executing clickjacking-like attacks is not possible in PROTECTION.

Replay attack. To prevent the replay attack, the remote server adds a random identifier (`id`) in the form specification alongside the signature. With this identifier, the server keeps track of the user input. When the server receives a form submission data, it first checks if the user submitted the same identifier before. In case of a collision, the server rejects the data.

Not rendering QR code. The host may deny sending the QR code over the HDMI channel. We consider this to be a denial of service and does not compromise integrity of the IO data.

Redirection. The attacker could redirect the user to a phishing website that renders visually identical UI to that of the legitimate website. Redirection attack cannot break the integrity of the input because a legitimate remote server always requires the signed input from the user. Without a valid signed specification, the IOHUB never renders an overlay or sign any input.

Malicious instruction on the screen. The attacker may put a malicious instruction/label on the untrusted part of the screen to influence user inputs. However, when the user starts interacting with the overlaid UI, the default focusing mechanism (Lightbox) highlights only the secure UI and hides the rest of the screen.

Replication of Lightbox. The attacker can replicate the lightbox on any part of the screen. However, this does not compromise the integrity of the input as the legitimate remote server only accepts signed input from the IOHUB.

Multiple HIDs. The attacker can emulate multiple HIDs to avoid the tracking of the mouse pointer. However, this attack is ineffective as the IOHUB only tracks the mouse pointer that is connected to it (over USB interface).

BadUSB. BadUSB [25] is out-of-scope of this paper as in the attacker model (Section III-A), we assume that all the IO devices that are connected to the IOHUB are trusted.

Mouse acceleration/updates. The attacker can change the mouse acceleration or provide erratic mouse updates on the screen. Such manipulations only cause the IOHUB to lose track of the mouse pointer and stop relaying the mouse signal to the host altogether. The IOHUB uses the acceleration parameters from the default `libUSB` driver to cope with the mouse acceleration. Hence, such manipulation does not affect security.

B. Confidentiality

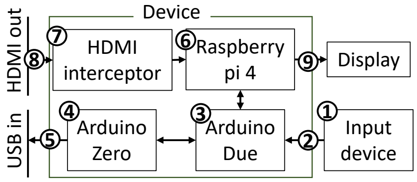
Redirection. The attacker could redirect the user to a phishing website that renders visually identical UI to that of the legitimate website. Redirection compromises the confidentiality of user inputs only when the user does not trigger the SAS mechanism. The IOHUB is only activated when it detects specifications signed from the whitelisted (maintained in the memory) servers.

Side-channel leakages. Even though, the IOHUB ensures that no mouse or keyboard event arrives at the untrusted host when the user executes some operation over the overlaid UI, one can not rule out all side-channel leakages. Depending on the application, the amount of time that the user spends or the entry/exit position of the mouse pointer may reveal some information to the attacker. IOHUB could allow the remote server to specify additional policies in the specification to prevent such side-channel attacks, e.g., a minimum amount of time that the device should not forward any event to the host after the user enters the overlay. We leave as future work defining such policies and integrating them on PROTECTION.

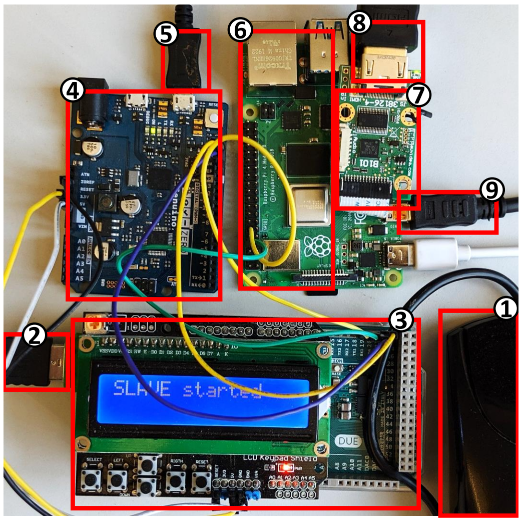
Mode Switching. The host could remove the QR code when the user is typing confidential data in the sensitive form. Absence of the QR code makes the IOHUB to assume that the secure session has ended and the IOHUB forwards the plaintext keystrokes and mouse movement to the host. To prevent the leakage of the input data, the IOHUB continues to overlay and operate on the overlay till the user clicks submit or cancel (or any UI element that has a `trigger` capability). This way, the IOHUB locks the UI from the attacker until the user finishes her session.

C. *Attacks toward* IOHUB

In PROTECTION trust model, we assume that the IOHUB is trusted. However, in real-world, embedded systems are often vulnerable to attacks as the attacker can use the connection



(a) The figure shows the basic components and connections between them in our PROTECTION prototype.



(b) The figure shows PROTECTION prototype that employs Arduino Due and Zero microcontroller board and a Raspberry Pi 4 SBC. The highlighted numbers correspond to the labels in Figure 9a.

Fig. 9: **PROTECTION prototype**. Figure 9a and 9b shows the component diagram and a photo of the actual PROTECTION prototype respectively.

interfaces to reprogram the IOHUB. As the code base of the IOHUB is small, we assume that the code can be formally verified to be protected against such attacks. However, we consider making a security-hardened IOHUB is engineering intensive and out-of-scope of this paper.

Downgrade attack. The host can block the initial QR code from the server to the IOHUB. By doing so, the host forces the server to downgrade the security of the webpage, i.e., not serving the PROTECTION JS. For integrity, this is not a security threat as the server does not accept any input from the host that is not signed by the IOHUB. Hence, the downgrade attack works as a denial of service which is out-of-scope of this paper.

VII. PROTECTION PROTOTYPE IMPLEMENTATION

Setup. Here, we describe our prototype implementation of PROTECTION as an auxiliary device. Figure 9 depicts the PROTECTION prototype in two parts: Figure 9a shows the block diagram of our prototype with various components and connections, and Figure 9b shows a photo of the actual prototype that highlights all the components described in the block diagram. The prototype IOHUB is connected to a desktop computer with 3.40 GHz Intel Core i7-6700 processor with 8 GB RAM running Ubuntu 18.04.2 LTS. The IOHUB uses off-the-shelf devices and has the following components (we use the same numbering as shown in Figure 9a and Figure 9b):

(i) Computing component. We use a Raspberry Pi 4 (⑥) to implement the computing component that executes all the IOHUB logic that includes analyzing the HDMI frames, rendering the overlays, executing the TLS protocol, etc. One could use an ASIC to further improve the performance and reduce the code base of the component. The Pi is connected to the display over HDMI (⑨) interface. The code base of the Pi primarily consists of Python and Java.

(ii) Input interceptor. The input interceptor is composed of an Arduino Due (③) and an Arduino Zero (④) that is connected to the input device over USB (②) interface. The input interceptor has a USB out interface that connects to the host (⑤) that relays all the user inputs to the host.

(iii) HDMI interceptor. The HDMI interceptor (⑦) is implemented using a B101 HDMI to CSI-2 Bridge [26] that takes the HDMI channel (⑧) from the host and convert it to the camera input signal to the Raspberry Pi 4.

A. Implementation of PROTECTION Components

In the following, we provide the implementation details of the PROTECTION components presented in the previous sections. Detailed implementation is provided in Appendix A.

*1) **QR code generation & UI specification:*** QR code generation phase is executed by PROTECTION JS that transforms the UI elements of a sensitive web form to a UI specification encoded in a QR code (we use QRCode.js, a JavaScript library to produce QR codes). Section IV-A provides the high-level concept of generating the QR code from the webpage UI elements. UI elements that require IO integrity protection can be marked by the developers in the HTML source. As illustrated in Figure 5, the HTML UI elements: '`Sensitive field 1`' and '`Sensitive field 2`' have the additional attribute `protect="true"`.

The PROTECTION JS iterates through the HTML elements that have the `protect` attribute enabled and extracts the information such as the name of the label or the type of the UI element. IOHUB uses preloaded size parameters to specify the size of a text field, button, etc. in case the size is not explicitly mentioned in the HTML source. One important attribute for a UI element in the specification is the `trigger`. For example, in Specification 1, the `OK` and the `cancel` buttons have an attribute `trigger`. This attribute is Boolean can be either `true` (corresponding to `OK`) or `false` (corresponding to `Cancel`) value. The value `true` denotes that the `OK` button can submit the values that are provided by the user. The `false` attribute denotes that hitting the `cancel` button abort the form altogether.

The QR code generation phase is between ① and ② in Figure 5 where the PROTECTION JavaScript snippet transforms the UI elements to a UI specification language in a QR code that can be interpreted by the IOHUB. The UI specification corresponding to the HTML source (in Figure 5) is provided in Specification 1. Note that the specification is highly flexible, allowing adjustable size for the form, individual UI elements, gaps between them, etc. This allows the IOHUB to faithfully recreate the UI that is very close to the actual form UI that the served by the web severer.

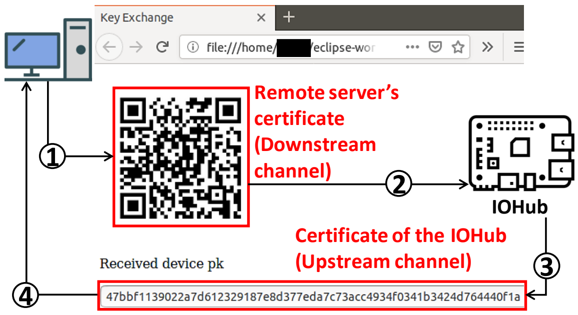


Fig. 10: **Establishing TLS.** A snapshot of the key exchange web page that is used to communicate the public certificates of the device and the remote server.

*2) **Bitmap generation:*** The IOHUB reads the QR code from the HDMI frame and generate the UI overlay bitmap from it. We have used the `piCamera` library to intercept the HDMI frames and generate the UI on top of it. Our PROTECTION prototype implements the most frequently used HTML input elements [27] that are common in sensitive forms.

*3) **Detection of mouse pointer:*** Initially, when the system boots up the IOHUB perform the calibration phase (see Section IV-C1) to synchronize its coordinates of the pointer with the host. The detection of the mouse pointer is implanted partially on the raspberry pi 4 (⑥ in Figure 9), while the mouse intercepting is done in the Arduino Due (③ in Figure 9). The Due gathers the raw mouse data (in terms of displacement measurements $(\Delta x_i, \Delta y_i)$) and sends them to the Pi over `Serial` interface. To guarantee that the IOHUB and the host interpret displacement events likewise, the Pi performs an adjustment operation. This operation consists of the IOHUB detecting the exact position of the host pointer in the HDMI frame by analyzing a small square of the frame (200 x 200 px) around its pointer coordinates. Considering that the IOHUB gets raw HDMI frames and pointer images are static, we use the lightweight `template matching` algorithm of the OpenCV library for the detection.

*4) **Implementation of the upstream channel:*** The *upstream* channel, i.e., the data from the IOHUB to the remote server is transmitted using the PROTECTION JavaScript snippet that is served by the remote web server. The PROTECTION JavaScript snippet uses a hidden text field to accept data coming from the IOHUB. The IOHUB emulates itself as a composite human interface device (HID) when it is connected to the host. The IOHUB emulates keystrokes that transmit encoded data (base64) to the PROTECTION JavaScript snippet that is sent to the remote server via `XMLHttpRequest` call.

*5) **Establishing TLS:*** For the IO confidentiality, the IOHUB and server create a TLS channel. When the user opens up a secure webpage, key exchange is the first step that takes place. We assume that the remote server already has the IOHUB's certificate, or some offline registration takes place. An instance of the key exchange protocol of PROTECTION is illustrated in Figure 10. The flow of the key exchange mechanism is as the following:

① The server delivers a web page with a QR code that encodes the signed public key of the server (server hello in TLS).

② The device captures every frame until it detects a QR code. Then, it decodes the QR code and verifies the public key and derives the shared secret using Diffie-Hellman protocol [28].

③ The device then sends its signed public certificate to the host, which forwards it to the server.

④ The remote server gets the signed certificate from the IOHUB, verifies it, and finally derives the shared secret.

VIII. PROTOTYPE EVALUATION

We evaluate the performance of our prototype by measuring the overheads introduced by PROTECTION to the system and whether they influence the user's interaction. Initially, we measure the default latency introduced by IOHUB when the user interacts with applications that do not require protection. Table II provides the relevant latencies. The delay in forwarding keystrokes is 170 μs and for frames is 21.76 ms. This allows the IOHUB to achieve the maximum display frame rate of 47.69 per second (e.g., most of the movies are shot ands shown in 24-30 fps). However, an optimized implementation of the technique to encode information in the HDMI frame would reduce significantly the processing time of a frame and increase further the frame rate as a result. Note that the B101 HDMI to CSI board that we use to intercept HDMI, has a hardware limit of 25 frames at 1080p resolution.

Our prototype of PROTECTION does not require the user to install any additional software in her machine to facilitate the communication between the remote server and the IOHUB. Instead, the IOHUB communicates with the remote server by using the host as an untrusted transporter. Therefore, we start by measuring the delay of sending data from the device to the host and vice versa:

IOHUB $\rightarrow$ host. The IOHUB transmits data (encrypted) to the host by simulating keystrokes. In our system IOHUB sends the keystrokes in a chunk of 256 bytes of data to the host. The keystroke has an average latency of 5 ms which is undetectable by humans.

Host $\rightarrow$ IOHUB. The host sends data to the device by encoding them into the HDMI frame. The QR-code is generated locally in the browser and displayed on the screen. For a specification of a form with two/four elements QR-code generation takes 14 ms. The IOHUB detects the QR-code, decodes it, and creates the overlay. This process takes 6 ms for the same form considered previously.

Initial Page Load. First time the user visits a web page that employs PROTECTION, the remote server and the IOHUB should exchange a cryptographic key to protect the communication. This step requires only one additional `xmlHttpRequest` to the server therefore the delay is relatively low. Initially, the browser encodes server's public key into a QR-code that is decoded by the IOHUB, which sends the response to the server by simulating the keystrokes.

Frame processing for mouse. IOHUB processes every frame of the host for pointer detection. This takes 1.76ms, which does not impact the frame rate.

Keystroke latency. The IOHUB intercepts all user's keystrokes and forwards them to the host or renders in the screen. When rendering on the screen, the latency is 170 μs.

Security Requirements		R4				R1 / R2			R1	R3a/b	
Category	Solutions	Trust Assumption				IO Security Features				Usability	
		Hardware		Software		Input			Output		
		Requires TEE	External trusted HW	Isolated API/Drivers	Hypervisor/OS	Keyboard	Pointer	Touch	Display	No SI	PnP
Hypervisor/OS-based	Shadowcrypt [29]			✔	✔		✔				✔
	Browser-based [30]			✔	✔				❏		✔
	InContext [16]				✔		✔				✔
	Overshadow [16]				✔						
	Virtual ghost [31]				✔						
	TrustVisor [32]				✔						
	Inktag [33]				✔						
	Splitting interfaces [34]				✔	✔			✔		
	SP^3 [35]				✔	✔					
Hypervisor/OS-based, TEE-based	SGX IO [4]	✔		✔	✔	✔					
	SchrodinText [36]	✔			✔				✔		
TEE-based	BASTION-SGX [37]	✔				✔					✔
	Slice [38]	❏									
	TrustOTP [39]	✔				✔			❏		✔
	VeriUI [40]	✔		✔		❏			❏		
	AdAttester [41]	✔		✔				❏	❏		
	TruZ-Droid [15]	✔		✔		✔			❏		✔
	TrustUI [42]	✔		❏				❏	❏		✔
	VButton [14]	✔		✔		❏		✔	✔		
TEE-based, External HW	CARMA [43]	✔	✔							✔	
	PROXIMITEE [11]	✔	✔	❏		✔				✔	✔
	Fidelius [9]	✔	✔	✔		✔			❏		
External HW	FPGA-based [44]		✔			✔			✔		
	IntegriKey [8]		✔	❏		❏				✔	✔
	Terra [45]		✔	❏							
	PROTECTION		✔			✔	✔	✔	✔	✔	✔

✔ requires/supports ❏ partially requires/supports

TABLE I: **Summary of existing trusted path solutions** by their trust assumptions, security features, and usability. A lower trust assumption, a high number of security features and high usability are desired from a trusted path solution. SI and PnP stand for security indicator and plug and play respectively. The table also categorizes the trust assumptions, IO security features and usability in-terms of the required security and functional properties that we list in Section II-C).

Operation	**Average time**
Detecting mouse pointer (A)	1.76 ms
Detection QR code (B)	14 ms
Decoding QR code + Overlay (C)	6 ms
Effective display latency $(A+B+C)$	21.76 ms
Mouse latency	250 μs
Keyboard latency	170 μs

TABLE II: **IOHUB performance**. The table shows the latency corresponding to a number of PROTECTION prototype operations.

	Projects	**LOC/Binary size**
Browser	Chromium (Google Chrome) [46]	25,163,547
	Mozilla Firefox [47]	20,928,358
JS Engine	Chrome V8 [48]	2,009,183
	Firefox SpiderMonkey [49]	2,908,550
OS	Ubuntu 18.10 (Debian)	1.9 GB
	Manjaro Linux 15.0.4 (Arch)	2 GB
IOHUB	HDMI interceptor + overlay	1,911
	USB stack	893
	Crypto stack	3,500
	RPi tiny core Linux	16 MB

TABLE III: **PROTECTION code-base comparison** with respect to some of the open-source browsers, JS engines and OSs.

Cursor latency. Similarly to keystrokes, the IOHUB intercepts mouse events also. However, the latency of event forwarding is 250 μs.

Codebase comparison. In Table III, we provide the code base and executable binary sizes of IOHUB with respect to some of the most popular open-source browsers, JavaScript interpreter engines and OS's. All of the codes are measured with the `cloc` open-source code line counting tool. The table shows that PROTECTION has significantly lower code base, resulting in a smaller attack surface.

IX. RELATED WORKS

In Table I, we summarize the existing research work based on their trust assumptions, IO security features, and usability. Note that it is desirable to have a lower trust assumption, higher security features, and higher usability. The trust assumption is further refined into hardware trust assumption that includes TEE and external trusted hardware, and software trust assumption, which includes isolated device drivers/APIs and trusted hypervisor/OS. The IO security features involve input that includes keyboard, pointer and touch input, and output that only includes the display. Lastly, the usability aspect is divided into two, the requirement of security indicator (SI), and if the solution supports plug-and-play (PnP). PnP implies that the solution can be integrated into the existing system without introducing any major changes into them and supports different architectures and OS out of the box.

Interpreting the table. The top of the table provides the required security and functional properties that are provided by PROTECTION. We list these properties in Section II-C. The trust assumption requires as minimum assumptions as possible (property R4). High number of IO security features are more desirable because of properties R1 and R2. The last category that is the usability of a system (in terms of low cognitive load on the users – R3a and R3b) can be improved if the security is not dependent on a security indicator, and the system provides a plug & play solution. Hence the systems with more entries in this category have better usability.

X. Conclusion

ProtectIOn provides a remote trusted path in the presence of an attacker-controlled host. The guiding principles behind our solutions are that (i) user input and output integrity cannot be considered separately, (ii) all user input modalities must be protected simultaneously, and (iii) user input integrity protection should not rely on user tasks that are prone to habituation and easily forgotten. By following these principles, we design a novel system that provide strong user input integrity protection in the presence of powerful adversary that controls the entire host platform.

References

[1] "X-600m | web enabled i/o controller," https://www.controlbyweb.com/x600m.

[2] "Inpen smart insulin delivery system | by companion medical." [Online]. Available: https://www.companionmedical.com/

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

[4] S. Weiser and M. Werner, "Sgxio: generic trusted i/o path for intel sgx," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017.

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

[6] T. Weigold and A. Hiltgen, "Secure confirmation of sensitive transaction data in modern internet banking services," in *2011 World Congress on Internet Security (WorldCIS-2011)*. IEEE, 2011.

[7] J. M. McCune, A. Perrig, and M. K. Reiter, "Bump in the ether: A framework for securing sensitive user input," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, Jun. 2006. [Online]. Available: /publications/papers/mccunej_bite.pdf

[8] A. Dhar, D.-Y. Yu, K. Kostiainen, and S. Capkun, "Integrikey: End-to-end integrity protection of user input," Cryptology ePrint Archive, Report 2017/1245, 2017, https://eprint.iacr.org/2017/1245.

[9] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia Jr, E. Gong, H. T. Nguyen, T. K. Sethi *et al.*, "Fidelius: Protecting user secrets from compromised browsers," *arXiv preprint arXiv:1809.04774*, 2018.

[10] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018.

[11] A. Dhar, I. Puddu, K. Kostianen, and S. Čapkun, "Proximitee: Hardened sgx attestation and trusted path through proximity verification," 2018.

[12] "Android protected confirmation: Taking transaction security to the next level," Oct 2018. [Online]. Available: https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html

[13] B. B. Anderson, A. Vance, C. B. Kirwan, J. L. Jenkins, and D. Eargle, "From warning to wallpaper: Why the brain habituates to security warnings and what can be done about it," *Journal of Management Information Systems*, 2016.

[14] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "Vbutton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018.

[15] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," 2018.

[16] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *SIGPLAN Not.*

[17] S. Egelman, L. F. Cranor, and J. Hong, "You've been warned: an empirical study of the effectiveness of web browser phishing warnings," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1065–1074.

[18] J. Sobey, R. Biddle, P. C. Van Oorschot, and A. S. Patrick, "Exploring user reactions to new browser cues for extended validation certificates," in *European Symposium on Research in Computer Security*. Springer, 2008.

[19] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning models," *arXiv preprint arXiv:1707.08945*, 2017.

[20] C. Sitawarin, A. N. Bhagoji, A. Mosenia, P. Mittal, and M. Chiang, "Rogue signs: Deceiving traffic sign recognition with malicious ads and logos," *arXiv preprint arXiv:1801.02780*, 2018.

[21] [Online]. Available: https://www.w3.org/TR/UISecurity/

[22] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 413–428.

[23] J. Lee, L. Bauer, and M. L. Mazurek, "The effectiveness of security images in internet banking," *IEEE Internet Computing*, Jan 2015.

[24] C. Marforio, R. Jayaram Masti, C. Soriente, K. Kostiainen, and S. Čapkun, "Evaluation of personalized security indicators as an anti-phishing mechanism for smartphone applications," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2858036.2858085

[25] "Badusb - on accessories that turn evil," 2014. [Online]. Available: https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf

[26] Admin, "B101 hdmi to csi-2 bridge (15 pin fpc)," Dec 2016. [Online]. Available: https://auvidea.eu/b101-hdmi-to-csi-2-bridge-15-pin-fpc/

[27] [Online]. Available: https://www.w3.org/TR/2012/WD-html-markup-20121025/

[28] S. Blake-Wilson and A. Menezes, "Authenticated diffe-hellman key agreement protocols," in *International Workshop on Selected Areas in Cryptography*. Springer, 1998, pp. 339–361.

[29] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "Shadowcrypt: Encrypted web applications for everyone," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1028–1039.

[30] Z. E. Ye, S. Smith, and D. Anthony, "Trusted paths for browsers," *ACM Transactions on Information and System Security (TISSEC)*, 2005.

[31] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," *ACM SIGARCH Computer Architecture News*, 2014.

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

[33] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[34] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.

[35] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008.

[36] A. A. Sani, "Schrodintext: Strong protection of sensitive textual content of mobile applications." in *MobiSys*, 2017.

[37] T. Peters, R. Lal, S. Varadarajan, P. Pappachan, and D. Kotz, "Bastion-sgx: Bluetooth and architectural support for trusted i/o on sgx," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '18. ACM, 2018.

[38] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms,"

in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011.

[39] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[40] D. Liu and L. P. Cox, "Veriui: Attested login for mobile devices," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 2014, p. 7.

[41] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015.

[42] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, p. 8.

[43] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. Van Doorn, "Carma: A hardware tamper-resistant isolated execution environment on commodity x86 platforms," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 2012.

[44] A. Brandon and M. Trimarchi, "Trusted display and input using screen overlays," in *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 2017.

[45] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*. ACM, 2003.

[46] Chromium, "chromium/chromium," Sep 2019. [Online]. Available: https://github.com/chromium/chromium

[47] Mozilla, "mozilla/gecko-dev," Sep 2019. [Online]. Available: https://github.com/mozilla/gecko-dev

[48] "V8 javascript engine." [Online]. Available: https://chromium.googlesource.com/v8/v8.git

[49] "Getting spidermonkey source code." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Getting_SpiderMonkey_source_code

[50] "picamera." [Online]. Available: https://picamera.readthedocs.io/en/release-1.13/

APPENDIX

PROOF FOR IO INTEGRITY

In this appendix, we provide a formal proof of the following property: *without protecting both input and output integrity, none of them can be achieved.*

A. Interaction Protocol

To simplify the proof, we model the interaction between the user, the host, and the remote server as a finite state automaton (FSA). The interactions between the server ($\mathcal{S}$), the user ($\mathcal{U}$) and host ($\mathcal{H}$) are depicted in the FSA in Figure 11.

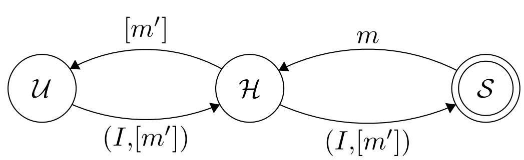


Fig. 11: Finite state machine that depicts the interaction between the user ($\mathcal{U}$), host ($\mathcal{H}$) and the server ($\mathcal{S}$).

$\mathcal{S}$ sends a message m to $\mathcal{H}$. One can assume m to be the HTML, JavaScript, and other data send from $\mathcal{S}$ as a HTTP response. We denote $[m]$ to be the render of m by the $\mathcal{H}$. As

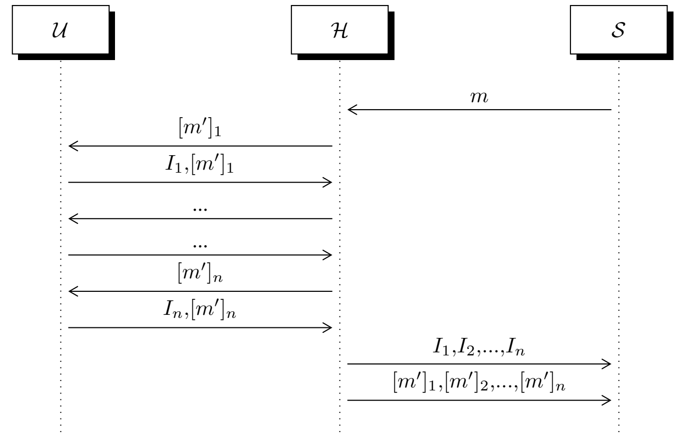


Fig. 12: Protocol transcript between the $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{H}$ that shows one trace from the FSM depicted in Figure 11.

$\mathcal{H}$ is malicious, it can transform m to m'. Note that the transformation is public knowledge and is deterministic. If $m \neq m'$ then given $[m]$ and $[m']$, $\mathcal{S}$ can determine that $[m] \neq [m']$. We denote the user input to be I, which corresponds to a specific $[m]$. In this model, we simplify the user input by assuming that the $\mathcal{U}$ only provides an input I only after observing a message transformation $[m]$. The user provides both her input I and transformation $[m']$ observed by her to $\mathcal{H}$. The interaction loop between $\mathcal{H}$ and $\mathcal{U}$ can continue until $\mathcal{U}$ finishes her input. After every input $\mathcal{H}$ hands over new message transformation to $\mathcal{U}$ (either result of the input or new message from $\mathcal{S}$ or both). Once the user provides all her inputs, $\mathcal{H}$ send the pairs $(I,[m'])$ to $\mathcal{S}$.

We also define two mappings:

$$\texttt{Input()} : [m] \rightarrow I$$

$$\texttt{Transform()} : m, I \rightarrow [m'],\ \exists i \in I : i = \phi$$

Both of them are *bijective*.

One trace of the protocol transcript is depicted in Figure 12. As described in the FSM, $\mathcal{S}$ receives traces of message transformation ($[m']_1$, $[m']_2$, ... , $[m']_n$) and corresponding inputs ($I_1, I_2, ..., I_n$). From these traces $\mathcal{S}$ could determine of all the $[m']_i$ are in proper form by verifying if $[m]_i = [m']_i$.

Definition A.1. Input integrity Assume that $\mathcal{S}$ handed a message m to $\mathcal{H}$ where the proper message transformation is $[m]$. The host changes the message transformation to $[m']$ where $[m'] \neq [m]$. We also define correct $\mathcal{U}$ input to be I when $\mathcal{H}$ sends a correct message transformation $[m]$ to $\mathcal{U}$. We define input integrity as the property where the $\mathcal{S}$ does not accept input I' where $I' \neq I$from $\mathcal{U}$ if the $\mathcal{H}$ changes the message transformation.

Definition A.2. Output integrity Assume that $\mathcal{S}$ handed a message m to $\mathcal{H}$ where the proper message transformation is $[m]$. Output integrity defines that in all circumstances, $\mathcal{U}$ receives the correct message transformation $[m]$ from $\mathcal{H}$.

Verification process. $\mathcal{S}$ checks $\forall i = 1...n$

$$[m']_i = \texttt{Transform}(m_{i-1}, I_{i-1})$$

where $I_0 = \phi$.

Theorem 1. *If $\mathcal{U}$ does not send all the transformations till $[m']_i$ corresponding to the input I_i, input integrity can not be* *achieved.*

Proof: If $\mathcal{U}$ does not attach all the transformation till $[m']_i$, i.e., $[m']_1$, $[m']_2$, ... , $[m']_{i-1}$, $[m']_i$ corresponding to inputs I_1,I_2,...,I_{i-1},I_i, then the server can not verify all the transformations corresponding to the input. $\mathcal{H}$ could modify a specific $[m]_x$ to influence $\mathcal{U}$ input. ■

Theorem 2. *If the channel from $\mathcal{U}$ and $\mathcal{S}$ is not authenticated, input integrity is not achievable. But the channel from $\mathcal{S}$ to $\mathcal{U}$ does not require to be secure as long a $\mathcal{U}$ provides the message transformation $[m']_i$ corresponding to every input I_i.*

Proof: The proof is trivial. If the channel from $\mathcal{U}$ to $\mathcal{S}$ is not authenticated, any input provided by $\mathcal{U}$ can be manipulated by $\mathcal{H}$ without a trace. Hence input integrity is not achievable. As long as $\mathcal{U}$ sends message transformation along with the input, a manipulated message transformation bt $\mathcal{H}$ would be detectable by $\mathcal{S}$ (see Theorem 1). ■

Theorem 3. *Ensuring output integrity also ensures input integrity provided there is an authenticated channel from $\mathcal{U}$ to $\mathcal{S}$.*

Proof: This proof is also trivial. As we describe in the Definition A.1 and A.2, if all the message transform from $\mathcal{H}$ $[m'] = [m]$, and $\mathcal{H}$ always executes `transform()` properly, the input integrity is preserved. As PROTECTIOn ensures output integrity and all the input from the user is signed by the IOHUB, PROTECTIOn preserves input integrity. ■

IMPLEMENTATION DETAILS

B. HID Drivers

We use Arduino prototype development board as the HID drivers. Figure 9b shows an Arduino Due, and a Zero board where the Due connects to the HIDs via the native USB port and the Zero relays the HID data to the Raspberry Pi (RPi). The Due and the Zero boards are connected over I^2C interface. As both Due and Zero only have one native USB port on each of them, we were forced to use two boards as an HID interceptor and relay. The Zero relays the HID signals both to the connected host (over native USB) and to the RPi (over serial interface). The connection from the Zero to the host is one way and emulates a composite HID. While the connection between the Zero and the RPi is bidirectional. The HID drivers are implemented using the native Arduino `keyboard` and `mouse` library. On the RPi, no HID drivers were needed as the RPi receives processed HID data from the Zero (for the pointer: displacement over x and y-axis and for keyboard, ASCII characters).

C. HDMI Interceptor, Relay and Overlay

The RPi along with the Auvidea B101 HDMI to CSI bridge, acts as the HDMI interceptor and relay. The B101 board converts HDMI signals from the host as a camera input (via the CSI interface) to the RPi. This allows the RPi to access the HDMI frames as a stream of JPEG frames. The HDMI out of the RPi acts as the relay that connects to the monitor. On the RPi, we use Picamera API [50] to access the HDMI frames. The B101 is capable of processing 25 frames at 1080p resolution. Hence, this is the hardware bottleneck

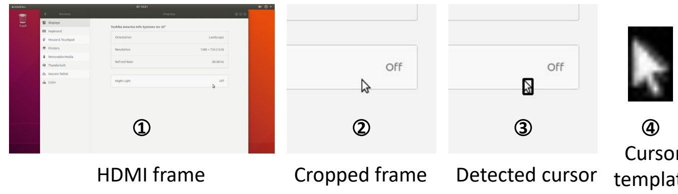


Fig. 13: **Cursor detection on the HDMI frame.** The figure shows PROTECTIOn mouse pointer tracking. ① shows the captured HDMI frame captured by B101 HDMI to CSI bridge. ② shows the cropped HDMI frame based on the mouse position received by the IOHUB. ③ shows the detected mouse pointer. For testing, we program the IOHUB to put a rectangle around the pointer. ④ shows one of the pointer templates that we used in our OpenCV routine.

of our implementation. However, the upcoming B112 board[1] could solve this performance issue. On the RPi, the overlay and HDMI out is implemented using Java SWT. Using SWT, we create a full-screen window that is shown on the monitor. The SWT class polls the HDMI frames and process them as individual JPEG images via the `BufferedImage` class. This allows the overlays to be drawn on the HDMI images efficiently. The Java program uses a QR code interpreter to extract the UI specification. Based on the UI specification, it creates the geometrical shapes (corresponding to the UI elements) and draw them on the frames. In the current implementation of the PROTECTIOn, the UI elements such as button, text-field, radio button etc. are preloaded in the IOHUB memory. Note that the current implementation of IOHUB is based on the RPi. But one could implement such functionality on an FPGA, reducing the TCB even more.

D. Mouse Pointer Tracking

The pointer tracing is also executed in the aforementioned Java program using simple object detection technique suppled by the OpenCV API. Figure 13 shows one screenshot of the pointer detection. The Figure shows the entire HDMI frame, the cropped frame of resolution 200×200 px (based on the mouse input data), the detected pointer in the cropped frame and the cursor template that is used by the object detection algorithm.

[1]still in development: https://auvidea.eu/showcase/.