IntegriKey: Integrity Protection of Keyboard Input for Security-Critical Web Services (Long Paper)

Anonymous Submission

Abstract. Many security-critical web services are used from hosts that are standard PCs. For example, industrial control systems, medical devices, and home automation systems are often configured through web interfaces. Similarly, online banking payments and cryptocurrency transactions are often executed through web interfaces. In such cases, the communication link from the host to the web server is typically easy to protect, but if the host platform gets compromised, the adversary can manipulate any user input with severe consequences, including safety violations and monetary loss.

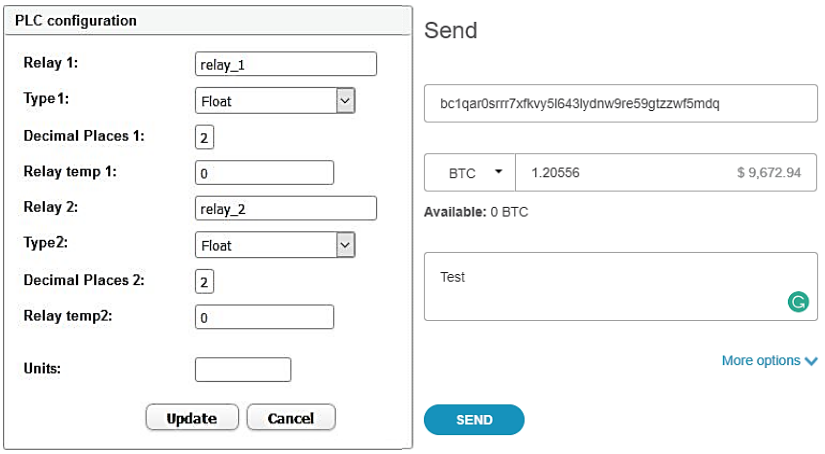
In this paper, we propose IntegriKey, a novel system for user input integrity protection in a compromised host. The user installs a simple plug-and-play device between the input peripheral and the host. This device observes user input events and sends a trace of them to the server that compares the trace to the application payload received from the untrusted host. To prevent subtle attacks where the adversary exchanges values from interchangeable input fields, we propose a labeling scheme where the user annotates input values. We built a prototype of IntegriKey, using an embedded USB bridge, and our experiments show that such integrity protection adds only a minor delay. We also developed a UI analysis tool that helps developers to protect their services. We evaluated our solution using the commercially available safety-critical system and online Bitcoin wallets as case studies.

Keywords: Input integrity · Web application security · Embedded systems

1 Introduction

Many web user interfaces implement security-critical functionality. Examples include web-based configuration of safety-critical devices like Programmable Logic Controllers (PLCs) used in manufacturing plants, medical devices, and home automation systems [10–12, 29]. Payments in online banking and cryptocurrency transactions from online wallets are additional examples of security-critical user interfaces implemented as web services. Figure 1 shows on the left a configuration web form for a commercial PLC system [15] that we use as a running example throughout the paper and on the right a browser-based Bitcoin wallet provided by BitGo [9]. The defining characteristics of such web services are that (i) they accept input from the user, (ii) the inputs are sensitive to minor changes, and (iii) after committing an input, the resulting action is difficult to reverse.

Typically in such web services, the communication between the host and the remote server (e.g., PLC, banking service, online wallet) is easy to protect through standard means such as a TLS connection [18]. However, if the host platform gets compromised—as standard PC platforms so often do—the adversary can manipulate any user-provided input. Such *user input manipulation attacks* are difficult to detect (before it is too late!) and can have serious consequences, including safety violations that can put human lives in danger and financial loss.



(a) (b)

Figure 1: Example configuration page. Screenshot from the (a) ControlByWeb x600m [15] I/O server configuration page, (b) web-based bitcoin wallet provide by BitGo [9]. In rest of the paper, we use the PLC server as the running example for our proposed system.

More generally, trusted input through an untrusted host platform to a remote server remains an open problem despite various research efforts [23, 26, 35, 36, 38, 39]. Indeed, all known approaches for *trusted input* have their limitations. For example, financial transaction confirmation from the display of a separate trusted device, like a USB dongle, is prone to user habituation and requires expensive additional hardware [35]. Secure input systems based on a trusted hypervisor have a large TCB and do not tolerate complete host compromise [36]. We review such prior solutions and their limitations further in Section 2.

Our goals and approach. In this paper, we address the *integrity protection* of user input in security-critical web user interfaces. Our goal is to design a solution that provides strong protection (e.g., no risk of user habituation, small TCB) and easy adoption (e.g., minimal changes to the existing systems and user experience, low deployment cost). We use remote configuration of a commercial safety-critical device (see Figure 1, left) as a running example, but we emphasize that our solution is not limited to that application domain. In Section 9 we discuss how the same approach can be applied to integrity protection of various other services including financial transactions, social media posts, and email.

The basic idea behind IntegriKey is straightforward and we call this approach *input trace matching*. When the user needs to perform a security-critical web operation like configuring a PLC device or performing a cryptocurrency payment, he installs a trusted embedded device between the user input peripheral and the host platform. This device intercepts user's input events, passes them through to the host, and sends a trace of them (over a secure channel) to an authenticated remote server that compares the trace to the user input received from the host to detect input manipulation. Once the user has completed the web transaction, he can remove the embedded device from the host to preserve the user's privacy and to prevent that any subsequent input events are sent to the same server unnecessarily.

This approach can be seen as a second-factor for user input integrity protection. If the primary protection mechanism (i.e., the integrity of the host platform itself) fails, the secondary protection provided by input trace matching ensures that the target safety-critical device cannot be misconfigured.

Secure and easy adoption of this idea involves overcoming some technical challenges.

The first is related to security, as an adversary that fully controls the host can execute restricted forms user input manipulation attacks, where he exchanges input values from interchangeable UI elements (e.g., two integers with overlapping ranges). Such *swapping attacks* cannot be detected by the server relying on the input trace alone. Another challenge is related to deployment. Our trusted device needs to communicate with the server, but we want to avoid building an (expensive) separate communication channel into it. We further want to avoid the need to install additional software on the host that could assist in such communication.

System and tool. Based on this idea, we design and implement IntegriKey, a user input integrity protection system, that is tailored for *keyboard* input, as keyboard input is sufficient for controlling many security-critical web interfaces including configuration of existing commercial safety-critical devices and execution of financial transactions.

Our system realizes the trusted embedded device as a simple USB bridge (for short Bridge) that is accompanied by a server-side user input matching library. To prevent subtle swapping attacks, our solution includes a simple *user labeling* scheme, where the user is asked to annotate interchangeable input elements. For easy adoption, we leverage the recently introduced WebUSB browser APIs to enable communication between Bridge and the server in a plug-and-play manner.

We also develop IntegriTool, a user-interface analysis tool that helps developers to protect their web services and minimizes the added effort of users. In particular, the IntegriTool detects input fields in web forms that require labeling and annotates the UI accordingly.

We implemented a prototype of Bridge using an Arduino board and evaluated IntegriTool using a range of existing web-based configuration UIs supported by x600m, a commercial PLC server [15], and several web-based Bitcoin wallets such as Bitgo [9], Bitcoin wallet [3], Coinbase [5], Coinspace [4] and Blockchain wallet [2]. Our results show that the tool can correctly process the configuration UIs of many existing security-critical web user interfaces. Our Bridge implementation adds a delay of 5 ms on the processing of keyboard events and its TCB is 2.5 KLOC.

We also conducted a preliminary user study where we simulated a swapping attack on 15 study participants. Labeling prevented the attack in 14 cases.

Contributions. To summarize, in this paper we make the following contributions:

- *New attack.* We identify swapping attacks as a novel form of user input manipulation against simple user input matching strategies.
- IntegriKey. We design and implement a user input integrity protection system that is tailored for keyboard input, prevents swapping attacks, and is easy to deploy.
- IntegriTool. We develop a user interface analysis and webpage annotation tool that helps developers to protect their web services and minimizes user effort.
- *Evaluation.* We verified that our tool can process UIs of existing safety-critical systems and cryptocurrency wallets correctly. Our experiments show that the performance delay of IntegriKey user input integrity protection is low. Our preliminary user study indicates that user input labeling prevents swapping attacks in most cases.

Outline. The rest of the paper is organized as follows. We explain our problem in Section 2. Section 3 introduces our approach, Section 4 describes our system and Section 5 the UI analysis tool. We provide security analysis in Section 6. Sections 7 and 8 explain our implementation and evaluation. In Section 9 we discuss other applications for our solution. Section 10 reviews related work and Section 11 concludes the paper.

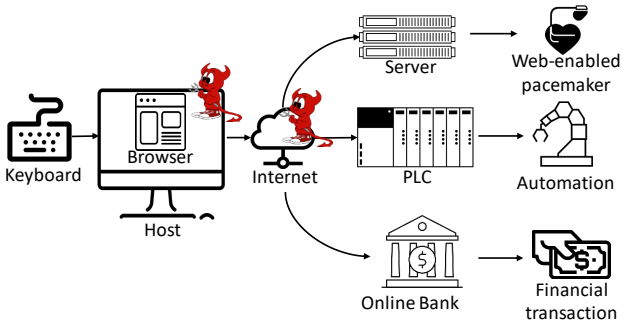


Figure 2: System model. We consider a setting where the user configures a safety-critical device or executes financial transaction through a web service from an untrusted local host. The user input device (keyboard) and the end-system (target device, bank server, medical implants etc.) are trusted. The host platform and the network connection can be controlled by the adversary.

2 Problem Statement

In this paper, we focus on the problem of user input manipulation by a compromised host PC in scenarios such as web-based remote configuration of safety-critical devices, financial transitions, emails and social media posts. (Attacks that compromise safety-critical systems directly are discussed in the literature; a survey of such works can be found in [20].)

2.1 System Model

Our system model is illustrated in Figure 2. We consider a common setting, where the user interacts with a security-critical web interface that, for example, configures a safety-critical device like a medical device, industrial robot or home automation system, or executes financial transaction over the Internet. The user provides input through an input device (in our case keyboard) to a web browser running on the *host* machine that is a standard PC. The browser sends the configuration input to a *server* that configures (or is) the safety-critical device.

We focus on *keyboard* input, as such input is sufficient to use the configuration web interfaces of many existing safety-critical devices [10–12, 15, 29] and online wallets [2–5, 9]. Later in the paper, see Section 9, we discuss the challenges of protecting other types of input devices, such as mouse input, with the same approach.

Adversary model. We consider the user input device (i.e., the keyboard) trusted and the target safety-critical device, and the server that receives the user input, also trusted. We assume that the adversary may have remotely compromised the host platform completely, i.e., the adversary controls the operating system, the browser, and any other software running on the host. We consider that even the host hardware can be exploitable. We assume that the adversary does not have physical access to the host platform.

We consider such strong adversary realistic, since OS vulnerabilities in PC platforms are well-known, browser compromise is increasingly common (see, e.g., [19, 32] for recent attack vectors) and hardware exploits are possible, e.g., through fabrication-time attacks [28, 37].

2.2 Limitations of Known Solutions

The problem of trusted user input to a remote server through an untrusted host has been studied in a few different contexts. Here we review the main limitations of known approaches, while Section 10 provides a more extensive review of related work.

Transaction confirmation. One common approach is transaction confirmation using a separate trusted device. For example, in the ZTIC system [35], a USB device with a small display and limited user input capabilities is used to confirm transactions such as payments. The USB device shows a summary of the transaction performed on the untrusted host and the user is expected to review the summary from the USB device display before confirming it. This approach is prone to *user habituation*, i.e., the risk that users confirm transactions without carefully examining them to be able to proceed with their main task, such as completing the payment, similar to systems that rely on security indicators [21, 22, 34]. Another limitation of this approach is that it breaks the normal workflow, as the user has to focus his attention to the USB device screen in addition to the user interface of the host. Finally, such trusted devices with displays and input interfaces can be expensive to deploy.

Trusted hypervisor. Another common approach is secure user input using a trusted hypervisor. Gyrus [26] and Not-a-Bot (NAB) [24] are systems where a trusted hypervisor, or a trusted VM, captures user input events and compares them to application payload that is sent to the server. SGXIO [36] assumes a trusted hypervisor through which the user can provide input securely to a protected application implemented as an Intel SGX enclave [8] which in turn can securely communicate with the server. The main limitation of such solutions is that even minimal hypervisors have large TCBs and vulnerabilities are often found in them [25, 31].

Dynamic root of trust. The third common approach is trusted user input using *dynamic root of trust* [30]. In the UTP system [23], the normal execution of the OS is suspended and a small *protected application* is loaded for execution. The protected application includes a minimal display and keyboard drivers, and is, therefore, able to receive input from the user and send it to the server together with a remote attestation that proves the integrity of the application handling the user input. The main drawback of this approach is that it changes the user experience of the web-based configuration application significantly, as small protected applications cannot implement complete web UIs. For example, the UTP system implements only a minimal VGA driver for text-based user interfaces.

2.3 Design Goals

Given these limitations of previous approaches, our solution has the following main design goals:

1. *Strong integrity protection.* Our solution should provide strong user input integrity protection even if the input host and the network are compromised. In particular, the solution should have a small TCB and not rely on tasks like transaction confirmations which are prone to user habituation.

2. *Easy deployment.* Our solution should be easy to deploy. In particular, we want to avoid significant changes to existing safety-critical systems, input devices, host platforms, or the web-based remote configuration user experience. We also want to avoid deployment of expensive hardware.

3 Our Approach

In this section, we introduce our approach and explain the technical challenges involved in realizing it in a manner that is both secure and easy to deploy.

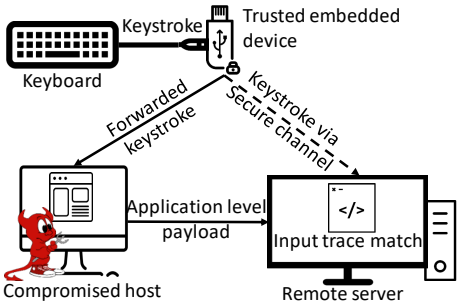


Figure 3: Approach overview. The user connects a trusted embedded device between the host and the keyboard. This device relays received keystroke events to the host and sends a trace of them over a secure channel to the remote server. The server compares the trace to the application payload received from the host to detect user input manipulation.

3.1 Input Trace Matching

We propose a conceptually simple approach for the protection of user input integrity that we call *input trace matching*. Our approach is tailored for keyboard input that is delivered to a remote server through a web application running on an untrusted host, as illustrated in Figure 3.

The main component of the solution is a trusted embedded device. When the user needs to perform a security-critical web transaction like configuring a PLC or performing a cryptocurrency transaction, the user connects the embedded device *between* the keyboard and the host. The connection from the keyboard to the embedded device and from the embedded device to the host can be wired (e.g., USB) or wireless (e.g., Bluetooth). We consider the embedded device trusted because it performs only very limited functionality and therefore it has significantly smaller software TCB, attack surface and hardware complexity compared to the host.

The trusted embedded device performs two types of functionalities. The first functionality is that it forwards received keystroke events from the keyboard to the host. The application running on the host (e.g., a web browser) receives the user input events and constructs an application-level payload (e.g., an HTTP response) that it sends to the server. Our approach imposes no changes to the host platform or the application software running on it. The second functionality of the embedded device is that it sends a trace of the intercepted keystrokes to an authenticated and authorized remote server over a secure channel when the user either changes the text field (by pressing tab key) or submits the form (by pressing an enter key).

The server parses the application payload received from the host and extracts the user input values from it. Then it compares input values to the received traces to detect any possible discrepancies. If the input values and keystroke events in the traces match, the user input can be safely accepted.

Once the user has completed the web transaction, he can remove the embedded device from between the host and the keyboard. This action prevent further input events from being forwarded to the server. Therefore, our solution does not violate user's privacy by exposing user's input outside the security-critical task to the authorized server. We discuss user privacy in more detail in Section 6.3.

3.2 Challenges

Realizing the above idea involves both security and deployment challenges that we discuss next.

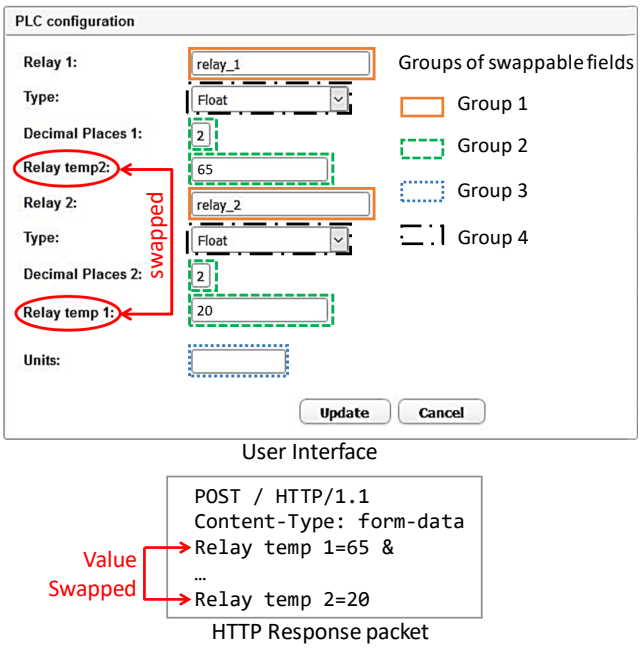


Figure 4: Swapping attack and interchangeable inputs. This screenshot shows our running example UI (PLC configuration web form), where the '`Relay temp 1`' and '`Relay temp 2`' user input field descriptions in the UI are swapped by the adversary. The corresponding HTTP response packet shows the swapped value of these two fields. Additionally, the figures shows groups of input fields that are swappable.

Swapping attacks. Input trace matching, as outlined above, prevents *most* user input manipulations by the untrusted host. For example, if the user types in one value, but the application payload contains another, the server can detect the mismatch and abort the operation.

However, the adversary may still perform more subtle and *restricted* forms of user input manipulation. The problem is exemplified by our running example UI, shown in Figure 4. Input trace matching allows the server to verify that all values received from the host were indeed typed in by the user, but since some values may *interchangeable* (i.e., they can have the same format and overlapping acceptable ranges), the untrusted host can perform a user input manipulation that we identify and call as *swapping attack*.

In a swapping attack, the malicious host manipulates the web form that is shown to the user and the application payload that is sent to the server. Figure 4 illustrates one such example, where the malicious host swaps the field names '`Relay temp1`' and '`Relay temp2`' in the UI. The user is likely to enter the values based on the swapped field names, but the server will interpret the user input differently, based on the manipulated HTTP response constructed by the host, as shown in Figure 4. Because the order of the user input in the received trace matches the HTTP response, the server cannot detect such manipulation from the input order. Assuming that the entered values are in the overlapping region of acceptable values for the respective input fields, the server cannot detect such manipulation based on the received values either.

Similarly, the adversary can swap any interchangeable user input fields (overlapping format and range) in the UI. Figure 4 shows a grouping for all interchangeable values in our example web form.

Adoption challenges. Input trace matching requires a secure communication channel from the trusted embedded device to the server. Our goal is to keep the device simple (small TCB) and inexpensive, and thus we avoid designs where the embedded device has its own communication capabilities (e.g., dedicated cellular radio). Host-assisted communication requires installation of new software on the host which can complicate adoption and in some cases may not even be possible for the user. Ideally, connecting the trusted embedded device to the host should be all the user has to do.

Another adoption challenge is that a single device should be able to provide user input integrity protection for multiple web services. The device can be configured with the keys and addresses of all supported servers, but during deployment, we want to avoid additional user tasks, such as manually indicating which of the pre-configured servers should be used.

4 IntegriKey: Input Protection System

In this section, we present IntegriKey, our system for user input integrity protection for remotely configurable safety-critical systems. Our system includes two main components: (1) the embedded trusted device realized as a simple USB bridge that we call for short Bridge and (2) a server-side user input matching library. To enable easy deployment, we use WebUSB [14], a recently introduced browser API standard supported by the Chrome browser. This API allows JavaScript code, served from an HTTPS website, to communicate with a USB device, such as our Bridge. To prevent swapping attacks, we propose a simple *user labeling* scheme where the user is instructed to annotate swappable input values.

4.1 Pre-configuration

Our system requires a secure (authenticated, encrypted and replay-protected) channel from the embedded device (Bridge) to the remote server. In our system, we leverage standard TLS and existing PKIs for this. To enable server authentication, the public key of the used root CA is pre-configured to Bridge. To enable client authentication, we use TLS client certificates. Each Bridge device is pre-configured with a client certificate before its deployment to the user, and the server is configured to accept such client certificates.

Besides input integrity protection, user authentication to the remote server without revealing the user's credential to the compromised host is also important. Our current implementation does not implement such user authentication, but in Section 9 we discuss how this can be enabled.

4.2 System Operation

Next, we describe the operation of the IntegriKey system that is illustrated in Figure 5.

1. The user attaches the Bridge device between the host and the keyboard. The user starts the browser on the host and opens the web page for remote configuration of the target safety-critical device.

2. The browser establishes a server-authenticated TLS connection (TLS_1) to the server.

3. The server sends the web configuration form to the browser together with JavaScript code. The web form includes instructions for user labeling, as described below in Section 4.3.

4. The browser shows the web form to the user and runs the received JavaScript code that invokes the WebUSB API to communicate with Bridge. The browser passes the server URL to Bridge.

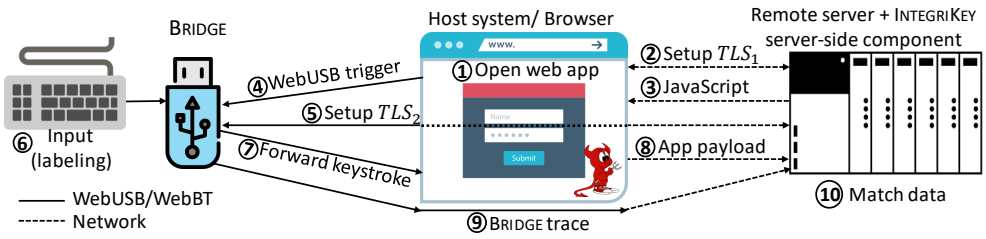


Figure 5: IntegriKey operation. The browser on the host opens a standard TLS connection (TLS_1) to the server which replies with a web page and JavaScript code. Using the WebUSB API, the JavaScript code invokes the BRIDGE that will establish another TLS connection (TLS_2) to the server. The BRIDGE forwards received keystroke events to the host and periodically sends a trace of them to the server that performs matching between the traces and the received application payload.

5. Based on the received URL and the pre-configured trust root and client certificate, BRIDGE establishes a mutually-authenticated TLS connection (TLS_2) to the server through the host using the WebUSB API. Mutually-authenticated TLS
6. The user completes the web form, as explained in Section 4.3.
7. BRIDGE captures keystrokes and forwards them to the browser.
8. Once the user has completed the web form, the browser constructs a payload (HTTP response) and sends this to the server over the TLS_1 connection.
9. BRIDGE collects intercepted keystrokes and periodically (e.g., when receiving a tab or return key press, or on every keyboard event) sends a trace of them to the server through the TLS_2 connection.
10. The server compares the received application payload and traces (input trace matching), as explained in Section 4.4. If no mismatch is detected, the server accepts the received user input. The user can remove BRIDGE from the host.

Handling multiple sessions/tabs. The WebUSB driver only allows one browser window to communicate with a USB device at a time. This restricts the BRIDGE to operate on multiple browser tabs or session at the same time. As INTEGRIKEY is targeted towards protection of specific security-critical web-based transactions (PLC configuration, payment) as opposed to generic input protection for all web browsing, we do not consider this a problem in practice.

Handing of key strokes. The BRIDGE acts as a USB host and handles all keyboard events from the user that includes modifiers such as `shift`, `ctrl`, navigation keys, character removal keys such as `backspace` and `del` etc. The BRIDGE registered itself as a generic USB plug-and-play device and emulates a keyboard. Hence, the BRIDGE replicates all the user keyboard actions and send the them to the browser along with the signed actions (traces) to the server. As one concrete example, assume that the user types `shift` + a, b, c and `backspace`, which appears as `Ab` in the browser. The BRIDGE records the trace as `[shift + a]+b+c+backspace` and translate to `A+b` which is received by the remote server.

4.3 User Labeling

To prevent swapping attacks explained in Section 3.2, we introduce a simple user labeling scheme. In this scheme, the user is instructed to annotate each interchangeable input with

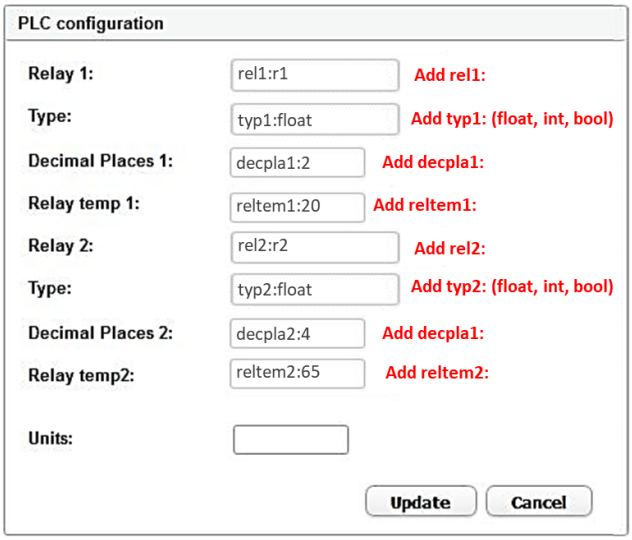


Figure 6: User labeling example. All input fields that need protection against swapping attacks are marked with labeling instructions. For example, to enter a value 20 to the input field `'Relay temp 1'` the user should type in `'reltem1:20'` as indicated in the web form next to the input field.

a textual label that adds *semantics* to the input event traces and thus allows the server to detect user input manipulation like swapping attacks.

An example of the user labeling process is illustrated in Figure 6. When the server constructs the web form, it adds labeling instructions to it. These instructions indicate the textual label, such as `'rel1:'` for input field named `'Relay 1'`, that the user should type in. The server adds such labeling instructions to each input field that needs protection against swapping attacks. In Section 5 we explain an automated UI analysis tool that helps the developer to securely find all such input fields and update the UI accordingly.

For each such field, the user types in the label followed by the actual input value. For example, to enter value `'r1'` to the input field with name `'Relay 1'`, the user types in `'rel1:r1'`. Some input fields may not require labeling. For example, the `'Units'` field in our example user interface (Figure 6) does not have to be labeled by the user as it is not swappable with any other field.

We consider trained professionals that configure industrial control systems, medical devices, etc. as the primary users of our solution. Such users can receive prior or periodic training for the above-described labeling process. The secondary target group is people such as home automation system owners. In this case, no prior training can be assumed, but the UI can provide labeling instructions.

4.4 Server Verification

To verify the integrity of the received user input, the server performs a matching operation shown in Figure 7.

Labeled inputs. First, the server parses through the application payload, and for every user input field that requires labeling makes the following checks: the server verifies that (i) the input appears in the expected position in the application payload, (ii) the input has the expected label, and (iii) one of the received input traces contains a matching labeled value. The order in which the correctly labeled value appears in the traces is not a reason

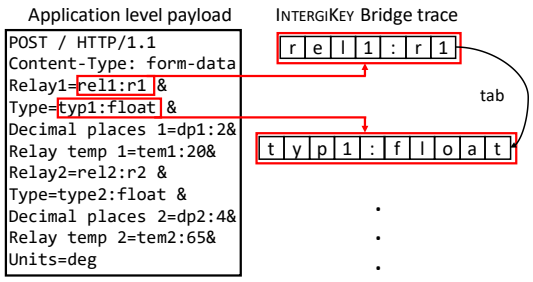


Figure 7: Input trace matching. The server compares user input values (and their labels) in the application payload (e.g., HTTP POST data) against the user input in the received traces.

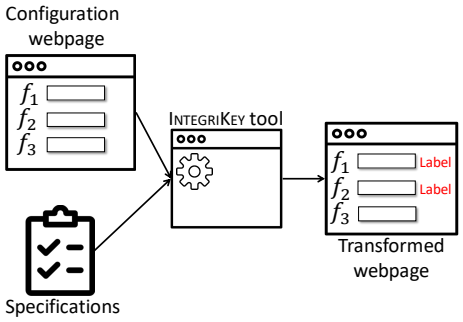


Figure 8: IntegriTool overview. IntegriTool takes a web page (HTML file) and a web page specification with input fields f_1, f_2, f_3. In this example f_1 and f_2 are swappable. The final output is a transformed webpage with labelling information (f_1 and f_2 requires labelling) for the users and converted mouse based UIs (drop-down menus, radio buttons, sliders etc.) to text fields.

for input rejection. For example, in Figure 7 the input labeled as `'rel1'` appears before the input labeled as `'typ1'`, but also the opposite order in the trace would be acceptable. Such a case might happen, if the user would fill in the web form values in an order that differs from the default top-to-bottom form filling.

Unlabeled inputs. Next, the server parses through the application payload again, and for every user input field that does not require labeling it performs the following checks: the server verifies that (i) the input appears in the expected position in the payload and (ii) one of the input traces contains the matching value. Also, unlabeled input values can appear in any order the in traces.

5 IntegriTool: UI Analysis Tool

Our labeling scheme helps web service developers to prevent swapping attacks. An obvious approach for developers is to require that users label all inputs. However, as labeling increases user effort, a better approach is to ask the user to label only those input fields that are interchangeable and thus susceptible to swapping. In this section, we describe a UI analysis tool, called IntegriTool, that helps developers to identify input fields that should be protected. When developers request labeling for those fields only, our tool also reduces user effort.

Specification 1: **Specification example.** This web page specification corresponds to our running user interface example that is illustrated in Figure 4.

```
<InputSchema>
 <Input>
   <ID>Relay 1</ID>
   <Format>s[a-zA-Z0-9]+[min=1,max>min]</Format></Input>
 <Input>
   <ID>Decimal places 1</ID>
   <Format>i[0-9]*[min=0,max=5]</Format></Input>
 <Input>
   <ID>Type 1</ID>
   <Format>m[{int, float, bool}]</Format></Input>
 <Input>
   <ID>Relay temp 1</ID>
   <Format>i[0-9]*[min=-20,max=150]</Format></Input>
 <Input>
   <ID>Relay 2</ID>
   <Format>s[a-zA-Z0-9]+[min=1,max>min]</Format></Input>
 <Input>
   <ID>Decimal places 2</ID>
   <Format>i[0-9]*[min=0,max=5]</Format></Input>
  <Input>
   <ID>Type 2</ID>
   <Format>m[{int, float, bool}]</Format></Input>
 <Input>
   <ID>Relay temp 2</ID>
   <Format>i[0-9]*[min=-10,max=100]</Format></Input>
 <Input>
   <ID>Unit</ID>
   <Format>s[unit][min=1, max=5]</Format></Input>
</InputSchema>
```

Figure 8 illustrates an overview of the tool that takes two inputs. The first input is the HTML code of the web form. The second input is a user interface specification that contains definitions for all input fields in the page. IntegriTool processes the provided inputs and outputs a generated webpage which is annotated with labeling instructions for the user. For example, for a user interface with input fields f_1, f_2, f_3 where f_1 and f_2 are interchangeable, the tool outputs a webpage with the labelling instruction for f_1 and f_2. An example screenshot of a webpage generated using our tool is shown in Figure 6.

5.1 UI Specification

The UI specification needs to be manually written by the developer. The specification captures the fact whether two user input fields in the UI are interchangeable. One such example is a home automation system, where the user can set the temperature of a specific room by providing the input to the web application. The attacker can swap input fields for temperatures of two rooms. Another and more interesting example is a UI where two fields are semantically different but share the similar format. Consider, for example, the configuration of a medical device, where the doctor can set blood pressure and heart rate limit. As the range of these two fields is overlapping, the attacker can swap the two fields even though they are semantically very different.

The user interface specification is an XML document that defines each user input field. Specification 1 shows an example for our running example UI. For each user input field, the specification provides the identifier of the web form element and its format. The format defines the type of the input (e.g., string (`s`) or integer (`i`)) and constraints for the acceptable value (e.g., a regular expression for a string or the minimum and maximum values for an integer). More precisely, we define the input format as:

$$type[regx][min = x, max = y][\{elements\}]^*$$

where *type* denotes the input field data type such as `string` (s), `integer` (i), `float` (f), `date` (d), `time` (t), `menu` (m) and `radio button` (r). $[regx]$ defines the regular expression for acceptable values. *min* and *max* define possible minimum and maximum `string` length or minimum and maximum values if the type is `integer`, `float`, `date` or `time`. The optional $[\{elements\}]$ is only applicable to UI objects such as `menu` (such as drop down menus) and `radio button`. $\{elements\}$ represents all the objects in the given UI element that can be chosen by the user.

We note that IntegriTool requires a tight specification to provide a precise output. If the developer provides a coarse-grained specification, that leads to an over-approximation of swappable fields by the tool that increases user effort but will not impose security risk.

5.2 Tool Processing

IntegriTool processes all input fields from the specification by evaluating them based on their specification. For numeric input fields (`integer`, `float`, `time`, `date`) the test checks for overlapping acceptable values, i.e., a boundary condition test. For `string` fields, our tool tests if the format constraints of two input fields can be met at the same time. For example, consider the following expressions:

$$RE_1 = s[a - zA - Z]^+[min = x, max = y]$$
$$RE_2 = s[a - zA - Z0 - 9]^+[min = x, max = y]$$
$$\implies RE_1 \subsetneq RE_2$$

where RE_1 represents a `string` containing uppercase or lowercase alphabetic characters and RE_2 represents a `string` containing uppercase or lowercase alphabetic or numerical characters. In this case, RE_1 is a subset of RE_2 as all strings from RE_1 are also members of RE_2 but there are strings in RE_2 that are not in RE_1. This can be verified by checking if $RE_1 \cap (RE_2)^c = \phi \implies RE_1 \subset RE_2$, where ϕ denotes empty set. In general, two fields f_i (corresponding regular expression RE_i) and f_j (corresponding regular expression RE_j) can be swapped if and only if $RE_i \cap RE_j \neq \phi$ and, f_i & f_j shares at least two elements. A short proof for this is as the following:

Proof. Let F_x and F_y be two input fields and their corresponding regular expressions are RE_x and RE_y. If F_x and F_y are swappable fields, then RE_x and RE_y have at least two overlapping accepted input.
If F_x and F_y are swappable, then

$$\exists x_i \in RE_x : x_i \in RE_y \text{ and } \exists y_j \in RE_y : y_j \in RE_x$$

This was the input values x_i and y_j can be swapped. Hence, $\{x_i, y_j\} \in RE_x \cap RE_y \Rightarrow |RE_x \cap RE_y| \geq 2$ □

Based on such tests, we design Algorithm 1 that generates a group of overlapping input fields. The algorithm works by *comparing every user input field to all the other fields* in the specification.

If one of the two compared fields is `string` and another is or number (`integer`, `float`, `date` and `time`) type, we check if their regular expression if overlapping (line 6). If one of the fields is `string` and another is either `menu` or `radio button`, then we check if an element of the `menu` (or `radio button`) is a member of the `string` regular expression (line 8). If both of the compared fields are of the numeric type, then we check for the boundary condition (line 13). The boundary check is also done for the elements of `menu` and `radio button` as the members could be number type (line 10). If both fields are `menu` or `radio button` type, then we check if the intersection of two fields is empty (line 14).

Algorithm 1: This algorithm finds swappable user input fields based on user interface specification.

Input: Specification S with input fields F.
Output: Set of subset of fields $G = \{g_1, \ldots, g_n\}$ where all the fields in a $g_i \in G$ are swappable.

$G \leftarrow$ Initialize empty group
for $\forall f \in F$ **do**
  for $\forall f_{in} \in F$ **do**
    $f.regEx, f_{in}.regEx \leftarrow$ read from S
    if $f.type =$ `string` **then**
      if $f.regEx \subset f_{in}.regEx$ **then** $addField \leftarrow true$
      if $f_{in}.type =$ (`menu` $\vee$ `radio button`) **then**
        if $f_{in}.elements \in f.regEx$ **then** $addField \leftarrow true$
    if $f.type =$ (`integer` $\vee$ `float` $\vee$ `time` $\vee$ `date`) **then**
      if $f_{in}.type =$ (`menu` $\vee$ `radio button`) **then**
        $f_{in}^{min} \leftarrow min(f_{in}.elements)$
        $f_{in}^{max} \leftarrow max(f_{in}.elements)$
      if $\neg(f^{max} < f_{in}^{min} \vee f^{min} > f_{in}^{max})$ **then** $addField \leftarrow true$
    if $f.type =$ (`menu` $\vee$ `radio button`) $\wedge f_{in}.type =$ (`menu` $\vee$ `radio button`) **then**
      if $f.elements \cap f_{in}.elements \neq \phi$ **then** $addField \leftarrow true$
    if $addField = true$ **then**
      $g \leftarrow$ empty set of fields
      $g.add(f, f_{in})$
      $G.add(g)$
      $addField \leftarrow false$
return G

Evaluating if a regular expression is a subset of another requires conversion of the regular expression to a deterministic finite automaton (DFA). The algorithm requires computing pairwise swappable tests over all the fields in the specification and returns groups of swappable fields. We analyze the complexity and performance of this algorithm in Section 8.

UI conversion. For drop-down menu and radio button inputs, our tool simply checks for overlapping `menu` and `radio button` elements. Our tool converts such elements into the corresponding textual representation to enable form completion with the keyboard. This is illustrated in Figure 9, where an example `radio button` with two options (`on` or `off`) is replaced with a textfield where the user is asked to type in either value `on` or `off`, correspondingly. Similarly, drop-down menus and slider elements are converted to a text input fields.

5.3 Web Page Annotation

The second output of IntegriTool is an annotated user interface. Our tool generates labeling instructions for users and embeds them into the web form, i.e., our tool instruments the HTML code. The instruction includes what label the user should add before each input value.

For choosing label names, we implement a simple approach, where IntegriTool takes the first three characters from each of the words. For example `'Relay temp 1'` converts to `'reltem1'`. Other label generation approaches are, of course, possible as well. In case

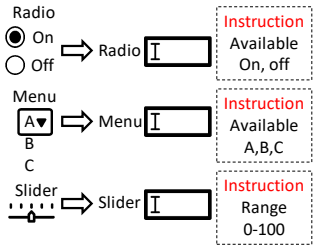


Figure 9: UI conversion. Conversion of radio button, drop-down menu and slider to an equivalent text field with added instructions.

of collision of generated labels, IntegriTool appends an incremented counter at the end of the label. Additionally, if there are multiple configuration pages (web forms) on the remote server that are identical, IntegriTool also appends an incremented counter. This ensures that no two text fields have identical labels.

An example of the tool's output is shown in Figure 6 which was produced using our running example UI and the specification listed in Specification 1 as inputs.

6 Security Analysis

In this section, we provide an informal security analysis of our system. The goal of the adversary is violate the *integrity* of web input. Examples include a misconfiguration of a safety-critical device or a false payment against the intention of the user.

6.1 Arbitrary Modifications

The simplest adversarial strategy is to manipulate only the application payload that is sent to the server. The adversary can, e.g., change one input value provided by the user to another arbitrary value in the HTTP response. Such attacks are detected by the server because the configuration data received over TLS_1 does not match the traces received over TLS_2.

6.2 Swapping Attacks

A more sophisticated adversarial strategy is to manipulate both the application payload and the user interface at the same time. More specifically, the adversary can change the names and the order of the user input fields and modify any instructions that are part of the user interface, such as the labeling instructions. Figure 4 shows one such example of the attack where the field names '`Relay temp 1`' and '`Relay temp 2`' are swapped.

The goal of a swapping attack is that the server interprets the received input values with different semantics than the user intended. Assuming that the user interface contains interchangeable fields, the adversary can construct an HTTP response where all input values are listed in the correct order and their values match to the input events. Two variants of such attacks are possible.

Manipulated field names or instructions. In the first variant, the adversary manipulates *either* the field names *or* the instructions. In such a case, the user interface that is shown to the user has an *inconsistency*, because the input field names and the labeling instructions do not correspond to each other. The user may react in different ways that we enumerate below:

- *Case 1: Abort.* The user may notice the inconsistency in the UI and abort the process.
- *Case 2: Correct labeling.* The user may perform the labeling correctly. That is, he may prefix each entered input value with the matching label. The target device is configured correctly, despite the user interface manipulation.
- *Case 3: Incomplete labeling.* The user may fail to complete the required labels. The server will abort the process.
- *Case 4: Incorrect labeling.* Finally, the user may perform the labeling incorrectly. That is, he may associate one of the asked input values with incorrect (and swappable) label prefix. The server cannot detect this case and the target device will be misconfigured.

In Section 8.3 we report results of a small-scale user study that provides preliminary evidence on how common these cases are, and especially how many people would fall for the attack (*Case 4*).

Manipulated field names and instructions. In the second variant, the adversary manipulates *both* the field names and the labeling instructions. In this variant, there are two possible cases. First, the labeling instructions do not correspond to the UI field order, in which case the effect is the same as above. Second, the modified labeling instructions correspond to the modified UI, i.e., the matching field names and labeling instructions are both modified the same way. In this case, the labeling instruction reordering essentially nullifies the effect of UI field name reordering and the UI is consistent again (no risk of misconfiguration).

We conclude that any manipulation of (*i*) UI field names, (*ii*) values in the application payload, (*iii*) labeling instructions in the UI, or (*iv*) the combination of thereof cannot violate input integrity unless there is a visible indication of it (i.e., inconsistency) in the user interface. This is the *Case 4* above which we evaluate with a small user study.

6.3 Privacy Considerations

In our solution, the trusted Bridge device intercepts the user's keyboard input events and sends a trace of them to the server for matching. As *any* interception-based solution has obvious privacy concerns, in this section we explain why the typical and recommended usage of our solution does not violate user's privacy.

1. *Device removal.* The primary usage model for our solution is one where the user attaches the Bridge device before a security-critical web operation and removes it after it. Thus, in such typical use, user input events outside the security-critical operation are not shared with the server. We do not recommend using our solution as a generic input protection mechanism for all web browsing, but rather as a hardening mechanism for only specific security-critical operations.
2. *Server white-listing.* The Bridge send the user input only to one (or more) pre-authorized (white-listed) servers. Therefore, even if the user would forget to remove the Bridge device from the host after the security-critical operation, the user input would be shared only with known and trusted servers. Such servers can implement additional privacy-preserving mechanisms like send a signal to the device to stop input sharing once the operation is completed.
3. *Safe handling of tabs.* As the Bridge uses WebUSB as the communication channel through the host's browser, the implementation of the WebUSB restricts only one

website (known as the *landing page*) to bind with the USB device [1]. This ensures that if the user switches the browser tab *during the security-critical operation*, the keystrokes from that application will not be forwarded to the Bridge unless the user reinitializes the device manually.

Finally, we emphasize that even in the worst case where the user forgets to remove the device and one of the white-listed servers turns out to be malicious, our solution does not *reduce* the user's privacy when compared to use without our solution. Since we assume a compromised host, the OS can trivially share all user input with any server regardless of whether our solution is used or not.

6.4 Other Security Considerations

Default values. IntegriKey eliminates any default values on the webpage. However, the host can always show default values to the user. If the user does not type the value by herself, the server rejects the input as the data from the browser and the trace does not match.

Trace dropping. Since all communication from Bridge to the server is mediated by the untrusted host, the adversary may also attempt to manipulate the traces by selectively dropping packets (e.g., remove certain user input). However, such attacks are prevented by the use of a standard TLS connection.

Cross-device attacks. An additional attack strategy is to trick the user to provide input for the configuration of one safety-critical device but use this user input for the configuration of another device. In such cross-device attacks, the host presents to the user the configuration user interface from server A but tricks Bridge to establish a connection with server B.

Cross-device attacks are only possible, if (*i*) the same Bridge is pre-configured for both servers A and B, (*ii*) every user input field in the configuration web pages of servers A and B is interchangeable, and (*iii*) both configuration pages have exactly the same labels. We consider such cases rare. To protect against cross-device attacks, both configuration user interfaces A and B can be processed with the same instance of IntegriTool which can annotate the pages with unique labels.

7 Implementation

We implemented a complete IntegriKey system. Our implementation consists of three parts: (1) Bridge prototype, (2) Server input trace matching library and (3) IntegriTool UI analysis tool.

7.1 Bridge Prototype

Our Bridge prototype consists of two Arduino boards and one USB switch, as shown in Figure 10. We used two separate boards, and an additional switch, because of the limited USB interfaces and computational power in the used Arduino boards, but we emphasize that a production device could be realized as a single embedded device In more detail, our Bridge prototype consists of an Arduino Leonardo board, a 16 MHz AVR microcontroller, which communicates with the host using WebUSB, and an Arduino Due, an 84 MHz ARM Cortex-M3 microcontroller, to execute computationally more expensive cryptographic operations needed for TLS. The two boards are connected using the I^2C protocol [16] in the master-slave configuration. The prototype can be connected to the keyboard via a

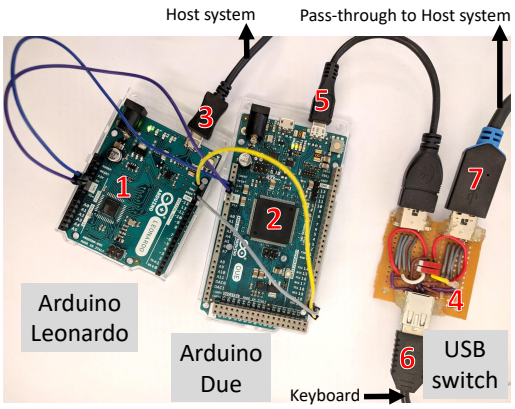


Figure 10: Bridge prototype. Bridge prototype consists of the following: 1) Arduino Due board is connected with the keyboard and executes cryptographic operations in TLS, 2) Arduino Leonardo board communicates with the browser using WebUSB, 3) USB connection from the Bridge to the host system, 4) USB switch to switch between the secure and insecure mode (pass-through), 5) the connection between the Bridge and the USB switch, 6) the keyboard connection, 7) the host pass-through connection for the insecure mode.

custom-made USB switch (see 4 in Figure 10). We use two boards as the WebUSB library we used only supported AVR boards such as Arduino Leonardo which is not powerful enough to execute cryptographic operations required by the TLS that we implement on the Arduino Due board. We already developed a limited version of the same prototype on a single Arduino Zero board using the newer driver to evaluate the viability of a single board.

As the currently available version of the WebUSB library [14] allows only one USB interface, our prototype cannot emulate a keyboard (interrupt transfer) and a persistent data (bulk transfer) device required for the TLS channel at the same time. Therefore, our prototype sends keyboard signals to the JavaScript code running in the browser. The JavaScript code interprets these signals and translates them to keyboard input on the web page. We use the Arduino cryptographic library for the TLS. The limited set of cipher suites in our TLS implementation uses 128-bit AES (CTR mode), Ed25519 & Curve25519 for signatures, Diffie-Hellman for key exchange and SHA256 for the hash function. Our prototype implementation is approximately 2.5K lines of code.

7.2 Server Implementation

Our server implementation for input trace matching is a Java EE Servlet hosted on an Apache Tomcat web server. We tested this implementation on a standard server platform, but the same code could be installed on a PLC server, such as [10–12, 15], as well. If a legacy PLC server does not allow installation of new code, our system could be deployed via a proxy, as discussed in Section 9. The server implementation consists of JavaScript that is served to the host's browser. We develop this JavaScript code that uses Google Chrome's WebUSB API to communicate with the Bridge. We use `XMLHttpRequest` to communicate with the remote server. Server users Java cryptographic library to implement TLS. The input trace matching is computed on the server after it decrypts the trace data from the TLS channel. This implementation is approximately 500 lines of code.

7.3 IntegriTool Implementation

We implemented the UI analysis tool in Java based on the Java AWT graphics library. The tool is around 1.5K lines of code and uses the Java native XML interpreter library to read the specification, dk.brics.automaton [6] for regular expression and *Jsoup* HTML parser to parse web pages.

8 Evaluation

In this section, we provide an evaluation of the IntegriKey system and the IntegriTool UI analysis tool. We also report results from a small-scale user study, where we simulated a swapping attack on 15 study participants.

Experiment setup. All experiments were performed on a laptop with a 3.2 GHz quad-core Intel i5 CPU and 16 GB memory running Ubuntu 16.10 64-bit. We used Google Chrome version 61 and JDK v1.8.

8.1 IntegriKey Performance

We evaluated the performance of our Bridge prototype using the following three metrics:

1. **Page loading latency:** The elapsed time between the web page load and when the Bridge is ready to take input from the user. The JavaScript code served by the remote server communicates with the Bridge and establishes a TLS using the WebUSB API. The additional TLS messages and the Bridge processing introduce this delay only at the initial loading of the page. We measure the difference between the time when the JavaScript code gets loaded into the browser and the time when the final TLS handshake message is sent.

2. **Keystroke latency:** The added processing delay when the user presses a key. This time is due to the internal processing of the Bridge. We place the measurement at program point where the `USBHost` library starts capturing the keyboard event and at the program point the device sends the data via the WebUSB interface to the browser.

3. **Communication overhead.** The communication overhead between the Bridge and the remote server. As the Bridge establishes a separate TLS channel with the server, this adds data overhead to the standard HTTP communication between the browser and the remote server.

We measured the page loading latency as 500 ms (includes loading of the WebUSB JavaScript and establishment of the TLS channel between the Bridge and the remote server) and the keystroke latency as 5 ms (both averaged over 500K iterations). These latencies are specific to the implementation architecture and the used boards, and that they can be reduced significantly using newer prototyping boards.[1]

Here, we emphasize that our solution is intended for user input integrity protection for specific security-critical operations. While a page loading latency of 500 ms may be significant performance penalty for many web applications like online commerce in general, such a delay is insignificant for the types of operations like configuration of safety-critical devices that we focus on.

[1] The master-slave I^2C channel is limited to 1 kHz. A Standalone implementation based on Arduino Genuino Zero, supported by the new version of the WebUSB driver eliminates the I^2C channel and reduces the latencies significantly.

Table 1: IntegriTool user interface processing time. We tested the processing time of our UI analysis tool on the web pages from the x600m PLC server, the home automation system and bitcoin wallets. All the measurements where conducted over 500K iterations.

Web page	#Fields	Processing time (ms.)	SD
x600m Web PLC			
Register configuration	6	1.654	0.0131
Counter configuration	7	0.771	0.0089
Event configuration	8	0.622	0.0085
Action configuration	5	1.241	0.0111
Supply voltage	4	0.673	0.0099
Calender configuration	11	0.713	0.0105
Home automation			
Home configuration	6	0.016	0.0018
Room configuration	5	0.012	0.0015
Bitcoin wallet			
Send Bitcoin	4	0.9	0.024

The communication overhead between the host system and the Bridge is very small. At the time of initialization, the Bridge and the remote server exchanges TLS handshake messages to establish the session. The handshake messages are 60 bytes each. Each TLS message adds extra overhead of 80 bytes (signature and announcement of next key).

We also tested the performance overhead of the server-side processing. The server has to maintain an additional TLS connection (TLS_2) which has a small cost and match the parsed HTTP response with the received user input events which takes less than a microsecond. From bandwidth point of view, the overhead of the second TLS channel is also small (this channel is only used to send the characters typed in by the user).

8.2 IntegriTool Evaluation

We evaluated our UI analysis tool implementation using two existing systems: PLC and home automation controller. The PLC system we used was ControlByWeb *x600m* [15] I/O server and we tested six separate configuration web pages for it. The home automation system we used is called home-assistant [7] and we tested two different configuration pages for it. We wrote UI specifications these pages and the fed the specifications to our tool implementation. The tool produced groups of interchangeable user input fields that we the manually verified to be correct. Table 2 in Appendix provides the details of this evaluation, including specifications for tested UIs and reported swappable elements. Based on this evaluation, we make two conclusions. The first is that our tool is able to process configuration UIs of existing, commercially-available safety-critical systems. The second is that many such UIs have swappable user input fields that need protection provided by our labeling scheme.

Additionally, we tested our UI analysis tool on user interfaces of other PLC controllers, home automation systems, medical device control, personal data management and online banking. Again, we wrote specifications for these user interfaces and processed them through our tool that founds swappable input elements in many of the tested user interfaces. We mined around 35 different text fields from 4 different application types. We list our findings in Table 3. We enumerate the user input fields in each page, their specifications that we manually created (including types and constraints) and the output of the tool that is grouping of swappable fields. We verified each output of the tool manually. We observe that in some cases the tool outputs as "swappable" input fields that can, in fact, be easily detected at the server. For example 'start date' and 'end date' are not swappable

as the former has to be less than the later. This is an example of a case, where both fields are specified correctly, but their relationship imposes additional constraints that can be checked by the server. For such type of fields, the developers can exclude them from the input specification.

Table 2: IntegriTool evaluation. We tested our implementation of the UI analysis tool on 'ControlByWeb x600m' industrial I/O server and 'home-assistant' home automation systems. Web pages column shows the configuration pages that we tested. We list types and formats for each user input field in the tested pages, and also list those input fields that are swappable.

Web pages	Fields	Type	Length/value constraint	Swappable fields
Web PLC configuration forms				
Register configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Type	`integer`	$[min = 1, max = 5]$	Units
	Units	`string`	$[min = 1, max = 5]$	Decimal place
	Decimal places	`integer`	$[min = 0, max = 5]$	Initial
	Initial Value	`integer`	$[min = 0, max = 999999]$	Type
Counter configuration	Device	`radio button`	{`on`, `off`}	Name
	Device counter number	`integer`	$[min = 0, max = 50]$	Description
	Name	`string`	$[min = 1, max = 20]$	Device counter number
	Description	`string`	$[min = 0, max = 60]$	Decimal places
	Decimal places	`integer`	$[min = 0, max = 5]$	Debounce
	Debounce	`integer`	$[min = 0, max = 9999]$	Edge
	Edge	`integer`	$[min = 0, max = 6]$	
Event configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Type	`menu`	{`int`, `float`, `boolean`, `constant`}	
	I/O	`menu`	{available IO}	
	Event group	`menu`	{available Groups}	
	Condition	`menu`	{`On`, `Off`, `Equals`, `Change state`}	
	Eval on powerup	`radio button`	{`yes`, `no`}	
	Duration	`integer`	$[min = 0, max = 9999]$	
Action configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Event source	`menu`	{available events}	
	Type	`menu`	{`On`, `Off`, `Toggle`,...}	
	Relay	`menu`	{Available relays}	
Supply voltage	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Decimal places	`integer`	$[min = 0, max = 5]$	
	Device	`menu`	{Available devices}	
Calender configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Event group	`integer`	$[min = 0, max = 5]$	Start date
	Start date	`date`	$[min = 01/01/2007, max = 31/12/2029]$	Stop date
	Stop date	`date`	$[min = 01/01/2007, max = 31/12/2029]$	Start time
	Start time	`time`	$[min = 00:00, max = 23:59]$	Stop time
	Stop time	`time`	$[min = 00:00, max = 23:59]$	Occurrence
	All day	`radio button`	{`on`, `off`}	Repeat val
	Repeat type	`menu`	{None, Secondly, Minutely, ...}	
	Repeat val	`integer`	$[min = 10, max = 9999]$	
	Occurrences	`integer`	$[min = 0, max = 999999]$	
Web Home automation configuration forms				
Home configuration	Room door lock	`radio button`	{`on`, `off`}	Room door lock
	Alarm	`radio button`		Alarm
	Water lawn	`radio button`		Water lawn
	Alarm time	`time`	$[min = 00:00, max = 23:59]$	
	Nest (thermostat)	`integer`	$[min = 16, max = 25]$	
	Sound selection	`menu`	{Available sounds}	
Room configuration	Table lamp	`radio button`	{`on`, `off`}	Table lamp
	TV back light	`radio button`		TV back light
	Celling lights	`radio button`		Celling lights
	AC	`integer`	$[min = 16, max = 25]$	
	Window shutter level	`integer`	$[min = 0, max = 10]$	

Example input fields. Table 3 provides provides a listing of additional web UIs that we analyzed using the tool. The list includes web pages for online banking page, medical programmer device, PLC server and home automation system. The main purpose of this table is to provide examples (or templates) for the developer for the fields which they are likely encounter while analyzing with IntegriTool. The table provides the names of the input fields along with their specifications, such as the regular expression and the length/value constraints, and whether some of the fields are mutually swappable or not.

We notice that some user input fields are strictly swappable only with another identical field. Such as an arbitrary field is not swappable with bank account number such as *IBAN* number due to the specific format (e.g., $[ISO3166-1$ IBAN code$][0-9A-Z]^+$ with minimum and maximum length of 20 and 30 respectively).

Table 3: Input field specifications. This table lists specifications (type, regular expression, length/value constraints) that we mined by analyzing various web pages (banking, medical, PLC, home automation). The swappable column denotes that the group of input fields with ✓ mark can be swapped with each other. Such as the current can be swapped with the frequency field.

Name	Type	Regular expression	Length/value constraints	Swappable
Personal information				
Email	`string`	$(*)^+(@)[a-zA-Z0-9]^+(.)[a-z]^+$	$[min=5, max=*]$	
Name		$[a-zA-Z.]^+$	$[min=1, max=*]$	
Address		$[a-zA-Z0-9]^+$	$[min=5, max=*]$	
Medical parameters				
Heartbeat	`integer`	$[0-9]^+$	$[min=55, max=210]$	✓
Blood pressure			$[min=80, max=150]$	
Blood sugar (Fasting)			$[min=108, max=126]$	
Body temperature	`float`	$[0-9]^+((.)[0-9])^*$	$[min=94, max=108]$	
Web-based PLC form [15]				
Analog Input(voltage)	`float`	$[0-9]^+((.)[0-9])^*$	$[min=0, max=12]$	✓
Current			$[min=300(mA), max=2(A)]$	
Thermocouple	`integer`	$[0-9]^+$	$[min=-15, max=150]$	
Frequency			$[min=0, max=500(Hz)]$	
Logic repetition			$[min=0, max=9999]$	
Event duration			$[min=0, max=9999999999]$	
Decimal places			$[min=0, max=5]$	
Initial value			$[min=0, max=999999]$	
Relay status	`radio button`	{On, Off}	$[min=0, max=1]$	✓
Thermocouple status				
Thermocouple status				
Energy slave status				
Input module status				
Thermostat status				
Logic start/end date	`date`	$[0-9]^+(/)[0-9]^+(/)[0-9]^+$	$[min = 1/1/2007, max = 12/12/2029]$	
Logic start/end time	`time`	$[0-9]^+(:)[0-9]^+$	$[min = 00:00:00, max = 23:59:59]$	
Logic Script	`string`	$(*)^+$	valid controller script	
Module name		$[a-zA-Z0-9]^+$	$[min=1, max=20]$	✓
Description			$[min=0, max=60]$	
Web-based home automation				
Room light toggle	`radio button`	{On, Off}	$[min=0, max=1]$	✓
Door lock toggle				
Alarm				
A/C				
Room temperature	`integer`	$[0-9]^+$	$[min=6, max=25]$	
Window shutter level			$[min=0, max=8]$	
Alarm time	`time`	$[0-9]^+(:)[0-9]^+$	$[min=00:00, max=23:59]$	
Web-based bitcoin wallet [2–5, 9]				
Address	`string`	$[1][P2PKH]^+$ $[1][P2SH]^+$ $[bc1][Bech32]^+$	$[min=34, max=42]$	
BTC	`string`	$[0-9]^+[.][0.9]^+$	$[min = 0.0, max = 9999999999.0]$	
Reference	`string`	$[0-9a-zA-Z.-\ ]^*$	$[min=0, max=*]$	
Financial transaction, online banking				
IBAN account no.	`string`	$(ISO3166-1$ IBAN code$)[0-9A-Z]^+$	$[min=20, max=30]$	
Transaction amount.	`float`	$(ISO4217$ currency code$)[0-9]^+((.)[0-9])^*$	$[min=0, max=*]$	

Processing time. We measured the processing time of our tool. Table 1 shows our results: the processing time of one web page varies from 0.01 ms to 1.65 ms. The processing time depends on the number of states in the DFA constructed from the regular expression of the specification and the number of input fields.

The time complexity of our UI analysis algorithm is exponential [33] ($\mathcal{O}(2^S)$) with respect to the number of states S in the non-deterministic finite automaton (NFA) that is derived from the regular expression that is quadratic $\mathcal{O}(|F|^2)$ with respect to the number of input fields $|F|$. In practice, the analysis of tested UIs was very fast as i) the number of input fields is usually 6 or less and ii) the DFAs from the specifications contain 2-3 states for most of the input fields.

8.3 Preliminary User Study

We also conducted a small-scale user study to understand whether the users can perform the proposed labeling correctly.

Recruitment. We recruited 15 study participants, aged 26-34, and all having a master's degree in computer science or related field.

Procedure. We prepared a web page extracted from the ControlByWeb x600m I/O server. We passed this page through our IntegriTool that annotated the page with the labeling instruction and converted drop-down menus to equivalent text fields. To simulate a swapping attack, we modified the page such that the description for the `'Relay temp 1'` and `'Relay temp 2'` fields were exchanged. The labeling instructions were unmodified.

We provided each study participant with an information sheet that provided brief background information on labeling and explained that the task is to configure a PLC device based on the provided instructions. We observed the study participants while they performed this task. Figure 11 shows the study UI and the information sheet.

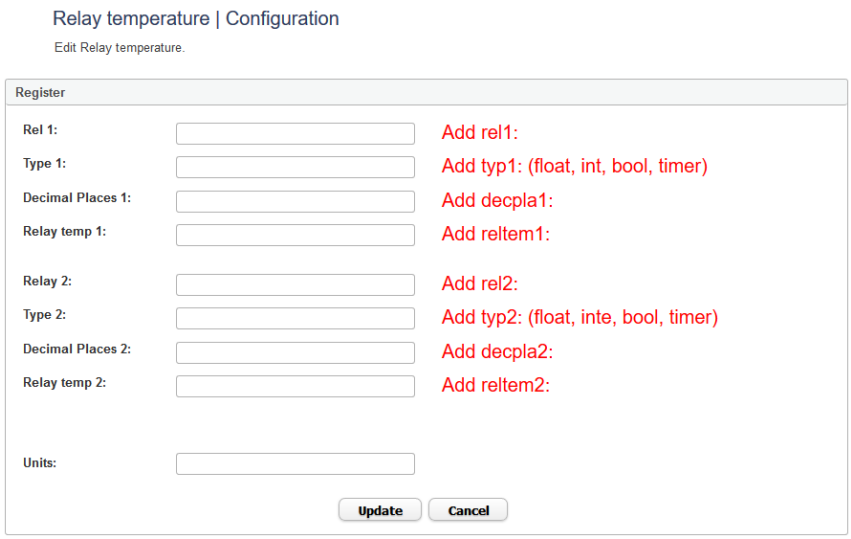
Results. Out of 15 participants, 7 noticed the inconsistency between the fields and labeling instructions in the UI, stopped the task, and report it to the study supervisor (*Case 1* in Section 6.2). Another 7 participants did not detect the UI inconsistency, but filled the input with correctly associated labels. The result is a correctly configured device (*Case 2*). One study participants completed the labeling incorrectly and fell for the attack (*Case 4*). The users reported that the additional labelling does not pose any significant effort. Additionally, none of the users made any mistakes while writing the labels in the text field along with the data.

Ethical considerations. Our user study did not collect any private information, such as email addresses or password. In such cases, our institution does not issue IRB approvals.

Study discussion. In our user study, we provided brief instructions the participants as shown in see Figure 11. This is in line with the primary usage of our system, where IntegriKey is used by trained professionals, who configure medical devices, industrial PLC systems and similar safety-critical devices. The secondary user group of our system is people like home automation system owners or cryptocurrency wallet users who have not received training for the task. Our study was not tailored for this scenario.

9 Discussion

Other application domains. In this paper we have focused on web-based configuration of safety-critical devices and payments through online banking and cryptocurrency wallets.



You have to fill a form that configures a remote safety-critical PLC. Misconfiguration may cause serious damage.
General instruction: Some of the input text fields require you to provide a label in front of the data. This prevent the host to manipulate the input parameters in case it is compromised. E.g., the abbreviated label for Relay temp 2 is reltem2. So, when you fill up the form, you write "reltem2:65"
You have to configure the following:

Relay 1= criticalRelay_1	Relay 2= criticalRelay_2
Type 1 = float	Type 2 = float
Decimal Places 1 = 2	Decimal Places 2 = 5
Relay Temp 1 = 20	Relay Temp 2 = 65

Figure 11: User study instructions. This figure shows the instruction sheet that was given to our user study participants.

However, our approach is not limited to those application domains. Additionally, one could integrate IntegriKey with browser-based email clients and social media to certify that the legitimate user is the one who composes the mail/post. This can be achieved by the Bridge to sign the mail/post and communicated directly to the server. On the receiving side, there can be two distinct scenarios: i) if the receiver has a Bridge installed on her host, the Bridge can check for the signature of the mail/post and validate it, ii) if the receiving side does not have a Bridge, the browser can use an extension to provide a secure indicator indicating that a legitimate Bridge indeed signed the mail/post.

Deployment. Assuming a browser that supports the WebUSB standard, our solution can be deployed without any changes to the host. The server-side component of our solution introduces small changes to the server. In case of legacy systems that are difficult to modify, the required server-side functionality could be implemented by a proxy server. Bridge could be configured to send the user input events to the proxy that could perform the input trace matching before passing the response to the unmodified legacy server.

Bluetooth. WebBluetooth [13] is another recent web API standard by Google Chrome that allows a JavaScript code to communicate with devices that are connected to the host. Our approach could be realized using WebBluetooth as well.

WebUSB security. The recent WebUSB and WebBluetooth APIs have received some criticism regarding possible security vulnerabilities. We emphasize that in IntegriKey, usage of WebUSB is not security critical, but it only enables communication from the Bridge to the remote server via the unchanged host. If the use of WebUSB should be avoided, IntegriKey can still be used with a browser plugin or an additional application on the host (that enables communication to the Bridge).

Other user input. Our current implementation is limited to keyboard input. To enable usage with various UIs with keyboard only, our tool converts elements, such as drop-down menus, sliders, and radio buttons, to text inputs. To extend our approach to pointer devices, such as the mouse, several aspects, such as mouse sensitivity and acceleration, and behavior of the mouse at the screen border would have to be considered. The fact that such mouse settings are controlled by the host OS would complicate the implementation. We investigated several commercially available PLCs, medical devices and online wallets, and learned that most of them can be configured through the keyboard alone.

User authentication. An adversary that controls the host is able to eavesdrop any user authentication credentials, such as passwords, entered to the host. To prevent such credential stealing, the trusted embedded device could be configured to act as an authentication token in addition to its main purpose of input integrity protection. For example, an administrator could configure the device with client certificates that could be used to authenticate the user during the establishment of TLS_1 connection to the server without revealing the authentication credentials to the untrusted host.

Secure autofill mechanism. Besides integrity protection of user-provided input, Bridge can be also used as a hardware-assisted autofill mechanism, eliminating the need for storing sensitive data such as user credentials, credit card number, etc. on the browser storage on the untrusted host system. The Bridge can use its internal flash storage to keep a key-value pairing of the identifier of the input fields and the actual data. The autofill operation is performed by the remote server to send the specific identifier of the input fields to the Bridge over the TLS channel between the Bridge and the remote server.

IntegriKey autofill has two phases: *i) Initialization* where the user provides input data to the web application for the first time. The flow of operation is identical to the standard IntegriKey operation described previously. The remote server sends the identifiers of the input fields to the Bridge. The Bridge stores the user data on its internal flash storage corresponding to the identifier. *ii) Autofill* phase allows the Bridge to populate the web forms in the browser from the data that are stored on its flash storage. The remote server achieves this by sending the identifier to the Bridge over the dedicated TLS channel.

Automated specifications. Our current implementation of IntegriTool requires that the developers specify the web page specification manually. An interesting direction for future work would be the development of a tool that parses the web page HTML and JavaScript code to generate the specification automatically.

Other channels. In our design, the connection from the trusted embedded device to the server shares the same physical channel as the browser, i.e., the Internet connectivity of the host. However, IntegriTool can be configured in such a way that this channel remains separated physically from the host. This can be achieved, for example, by using a smartphone application in the role of the trusted device. The drawback is increased TCB size.

Other trust models. We designed our system considering an adversary that can fully compromise the host. An alternative trust model, similar to [24, 26], would be one where the host OS trusted, but the browser, or one of its extensions, is compromised. Under such trust model, the OS could take the role of the trusted embedded device.

10 Related Work

The problem of protecting integrity of user input that is delivered to a remote server via an untrusted host has been studied previously in a few different contexts. Here we review the most related prior works.

10.1 User Intention Monitoring

The first set of related solutions focus on user *intention*. These systems attempt to ensure that the data received by the remote server is constructed as the user intended.

Gyrus [26] records user intentions, in the form of text input typed by the user, and later tallies it with the application payload that is sent to the server. On the host, Gyrus assumes an untrusted guest VM (dom-U) that can manipulate user input and a trusted VM (dom-0) that draws a secure overlay and captures the user input. The overlay is application-specific and covers critical input fields such as the website address bar, mail compose window etc. When the application sends a message to the server, dom-0 matches the captured user input data with the application payload.

Not-A-Bot (NAB) [24] attempts to ensure that data received from the host was generated by the user and not by a malicious software. Also NAB relies on a trusted hypervisor that loads a simple *attester* application whose software configuration can be verified through remote attestation. The attester records user input events and provides a signed statement of them to the server. Binder [17] is another similar system where a trusted OS correlates outbound network connections with the recorded user inputs events. The main difference between these solutions and our work is that we assume a fully compromised host.

10.2 Trusted Path Solutions

User input integrity has been studies also in the context of hardware-based trusted execution environments (TEEs). The term *trusted path* refers to a secure communication channel between the user and a protected application running on an untrusted platform.

UTP [23] describes a unidirectional trusted path from the user to a remote server using dynamic root of trust based on Intel's TXT technology [30]. The system suspends the execution of the OS and loads a minimal protected application for execution. This loading is measured and stored to a TPM and proved to a remote verifier using remote attestation. The protected application creates a secure channel, records user input and sends them securely to the server. The main drawback of this approach is that such minimal protected applications cannot implement complex (web) user interfaces. For example, UTP is limited to VGA-based text UIs to keep the TCB small.

SGXIO [36] assumes a trusted hypervisor and trusted device drivers and uses them to create a secure channel from the user to an SGX enclave. Intel's Software Guard Extensions (SGX) [8] is a trusted execution environment (TEE) implemented as a specific execution mode in the processor. SGX allows isolated execution of small protected applications (enclaves) and protects their secrets and execution integrity from any untrusted software running on the same platform. The main difference to our work is the need for a trusted hypervisor.

Zhou et al. [38] realize a trusted path for TXT-based TEEs, again relying on a small trusted hypervisor. In this solution, also device drivers are included in the TCB. Wimpy kernel [39] is a small trusted kernel that manages device drivers for secure user input. We, in contrast, assume a completely compromised host.

10.3 Transaction Confirmation Devices

The third set of known solutions use a separate trusted device to confirm user input for transactions like online payments. ZTIC [35] is a small USB device with a display and user input capabilities. This device shows a summary of the transaction performed on the untrusted host and the user is expected to review the summary from the USB device display before confirming it. Kiljan et al. [27] propose a similar transaction confirmation device.

Such solutions have three main drawbacks. First, they are prone to user habituation, i.e., the user will not always carefully review the transaction. Second, they break the normal workflow, as the user has to focus his attention to the USB device screen in addition to the normal UI on the host. Third, such devices can be expensive to deploy. Our solution is cheap to deploy and the user experience remains mostly unchanged.

11 Conclusion

Remote configuration of safety-critical systems is prone to attacks where a compromised host modifies user input. Such attacks can have severe consequences that can put human lives in danger. In this paper we have proposed a new solution, called IntegriKey, to prevent user input manipulation by the untrusted host. In our scheme, the user installs a simple embedded device between the user input peripheral and the host. This device sends a trace of user input events to the server that can detect input integrity violations by comparing it to the received application payload. Our evaluation shows that IntegriKey is cheap to build, easy to deploy, and it works in practice.

References

[1] “Access usb devices on the web | web | google developers.” [Online]. Available: https://developers.google.com/web/updates/2016/03/access-usb-devices-on-the-web

[2] “Bitcoin wallet - store and invest in crypto.” [Online]. Available: https://www.blockchain.com/wallet

[3] “Bitcoinwallet.com.” [Online]. Available: https://bitcoinwallet.com/

[4] “Coin.” [Online]. Available: https://coin.space/

[5] “Coinbase - buy/sell digital currency.” [Online]. Available: https://www.coinbase.com/

[6] “dk.brics.automaton - finite-state automata and regular expression for java,” http://www.brics.dk/automaton/.

[7] “Home assistant demo,” https://home-assistant.io/demo/.

[8] “Intel sgx homepage,” https://software.intel.com/en-us/sgx.

[9] “Making digital currencies usable for business.” [Online]. Available: https://www.bitgo.com/

[10] "Modicon momentum," https://www.schneider-electric.us/en/product-range/535-modicon-momentum.

[11] "Simatic s7-1200," https://www.siemens.com/global/en/home/products/automation/systems/industrial/plc/s7-1200.html.

[12] "Sitraffic smartguard," https://www.siemens.com/global/en/home/products/mobility/road-solutions/traffic-management/strategic-management-and-coordination/centrals/smartguard.html.

[13] "Web bluetooth," https://webbluetoothcg.github.io/web-bluetooth/.

[14] "Webusb api," https://wicg.github.io/webusb/.

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

[16] "I2c," https://learn.sparkfun.com/tutorials/i2c, 2018, accessed on 27.04.2018.

[17] W. Cui, R. H. Katz, and W. tian Tan, "Design and implementation of an extrusion-based break-in detector for personal computers," in *ACSAC'05*.

[18] T. Dierks, "The transport layer security (tls) protocol version 1.2," 2008.

[19] T. Dougan and K. Curran, "Man in the browser attacks," *International Journal of Ambient Computing and Intelligence (IJACI)*, 2012.

[20] C. Fachkha, E. Bou-Harb, A. Keliris, N. Memon, and M. Ahamad, "Internet-scale probing of cps: Inference, characterization and orchestration analysis," in *NDSS*, 2017.

[21] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo, "Rethinking connection security indicators," in *SOUPS 2016)*.

[22] A. P. Felt, R. W. Reeder, H. Almuhimedi, and S. Consolvo, "Experimenting at scale with google chrome's ssl warning," in *CHI*, 2014.

[23] A. Filyanov, J. M. McCuney, A. R. Sadeghiz, and M. Winandy, "Uni-directional trusted path: Transaction confirmation on just one device," in *IEEE/IFIP DSN 2011*.

[24] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-bot: Improving service availability in the face of botnet attacks," in *NSDI 2009*.

[25] K. Hashizume, D. G. Rosado, E. Fernández-Medina, and E. B. Fernandez, "An analysis of security issues for cloud computing," *Journal of internet services and applications*, 2013.

[26] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A framework for user-intent monitoring of text-based networked applications." in *NDSS*, 2014.

[27] S. Kiljan, H. Vranken, and M. V. Eekelen, "What you enter is what you sign: Input integrity in an online banking environment," in *2014 Workshop on Socio-Technical Aspects in Security and Trust*.

[28] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, *Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering*.

[29] B. Mahato, T. Maity, and J. Antony, "Embedded web plc: A new advances in industrial control and automation," in *2015 Second International Conference on Advances in Computing and Communication Engineering*.

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

[31] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 2013 international workshop on Security in cloud computing*.

[32] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, N. Modadugu *et al.*, "The ghost in the browser: Analysis of web-based malware." *HotBots*, 2007.

[33] K. Salomaa and S. Yu, *NFA to DFA transformation for finite languages*, 1997.

[34] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *S&P'07*.

[35] T. Weigold and A. Hiltgen, "Secure confirmation of sensitive transaction data in modern internet banking services," in *WorldCIS 2011*.

[36] S. Weiser and M. Werner, "Sgxio: Generic trusted i/o path for intel sgx," ser. CODASPY '17.

[37] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *S& P 2016*.

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

[39] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *S&P 2014*.