Hacking in the Blind: (Almost) Invisible Runtime User Interface Attacks

Luka Malisa, Kari Kostiainen, Thomas Knell, David Sommer, and Srdjan Capkun

Department of Computer Science,
ETH Zurich
{firstname.lastname}@inf.ethz.ch, knellt@student.ethz.ch

Abstract. We describe novel, adaptive user interface attacks, where the adversary attaches a small device to the interface that connects user input peripherals to the target system. The device executes the attack when the authorized user is performing safety-, or security-critical operations, by modifying or blocking user input, or injecting new events. Although the adversary fully controls the user input channel, to succeed he needs to overcome a number of challenges, including the inability to directly observe the state of the user interface and avoiding being detected by the legitimate user. We present new techniques that allow the adversary to do user interface state estimation and fingerprinting, and thus attack a new range of scenarios that previous UI attacks do not apply to. We evaluate our attacks on two different types of platforms: e-banking on general-purpose PCs, and dedicated medical terminals. Our evaluation shows that such attacks can be implemented efficiently, are hard for the users to detect, and would lead to serious violations of input integrity.

1 Introduction

Modern malware can reside in various places — on the device itself, but also on connected peripherals (e.g., BIOS or hard drive). One type of such malware resides in user interface devices, such as a keyboard or a mouse [19]. The goal of such malicious peripherals is to inject pre-programmed sequences of user input that, e.g., install some form of malware to the device itself. However, installing malware or introducing system misconfigurations, such as adding an administrative account, can be prevented by security hardening (e.g., Windows Embedded Lockdown features [16]), or by existing malware-detection approaches.

A significantly stealthier alternative is to attack systems *only* through their user interfaces (UIs). Such attacks exploit a fundamental property of any device designed to operate under user control; irrespective of applied security hardening techniques, the device *must continue* to accept user input.

As user input cannot be simply blocked, various UI attacks have been proposed. Current state-of-the-art are BadUSB-style attacks [19], where the goal of the malicious peripheral is to inject simple, *pre-programmed* sequences of keyboard and mouse input [11], commonly while the user is not active. However, due to the lack of system state awareness, such approaches are restricted to executing only simple attacks (e.g., add new account, install malware). For example, the BadUSB malware does not know in which state the system is currently in, and launching such attacks that inject user input at the wrong time could result either in attack failure, or in users trivially detecting them.

We therefore pose the following question: *"Can an adversary improve the state-of-the-art UI attacks, and expand the set of applicable attack scenarios?"*. We observe that if the malware could infer the system state, and the precise attack launch time, stealthy and more powerful attacks become possible.

We present a new class of adaptive runtime user interface attacks, where the adversary infers the system state, violates the integrity of user input at a specific point in time, *while the device is operated by the legitimate user*, causing precise and stealthy runtime UI attacks without any malware running on the device. Contrary to existing works, our attack does not blindly inject input events, but rather hijacks the input channel of a currently active user, and enables new attack scenarios (e.g., compromise integrity of e-banking payment) that are not achievable with existing UI attacks. The attack is hard for the user to detect, it can result in serious safety or security violations, and the users are led to believe that they accidentally caused the damage themselves.

The first part of the attack is conventional. The adversary gains temporary physical access to the target system and attaches a small attack device (e.g., similar to the NSA cottonmouth [1]) in between an input device and the system.

In the second part of the attack lies its novelty. Contrary to existing approaches, our attack device observes the constant stream of user input events and, based on this information, determines when the user is about to perform a critical operation, and when the UI attack should be launched. Although the attack device has full visibility and control of the user input channel, it *cannot directly observe the state of the target system*, as the device has no system feedback (e.g., access monitor output). In particular, the adversary does not know the current state of the UI or the mouse pointer location, and must therefore, given user input, infer the most likely system state and correct attack timing.

To successfully realize our UI attacks, we had to overcome technical challenges. Once the adversary has determined the correct time to attack, the attack device injects a series of precise and fast input events. While the adversary is able to freely manipulate the input channel, the user receives instant visual feedback — the legitimate user is part of the system control loop. Therefore, we designed novel attack techniques, including state tracking and fingerprinting, that are both accurate (low false positives), and stealthy (give little visual indication to the user). To demonstrate the attack, we implemented it on a small embedded device, and evaluated it on general-purpose PCs and medical terminals.

On PC platforms, we tested our attack on UIs of real-world e-banking websites. We can accurately fingerprint UIs in a reliable (90% attack success rate) and stealthy (90% of users did not notice our attack) manner, that is surprisingly tolerant to noise (users habitually clicking around, pressing "tabs" to navigate between elements, different browsers and screen resolutions). For dedicated ter-

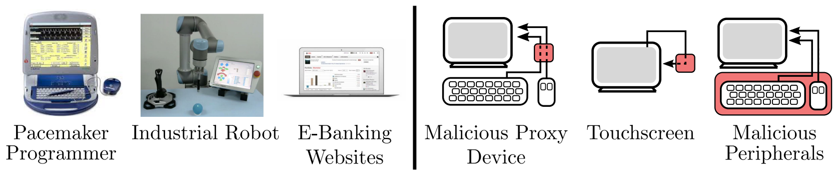


Fig. 1. (Left) Examples of critical user interfaces on dedicated terminals and general-purpose PCs. Attacking such UIs can lead to various safety and security violations. (Right) The adversary attaches an attack device to an user input interface.

minals, we tested the attack on a simulated UI of a medical implant programmer. We show that we can accurately perform system state tracking, and that the attacks are very hard to detect (93–96% success rate).

Performing such attacks is not possible using existing UI attacks, *unless* the malware resorts to injecting malicious software onto the target device; a step that we never resort to. We emphasize that our attack approach is applicable to a wide range of attack scenarios, including different user input methods and UIs, where the adversary has to perform the attack under target state uncertainty. Our attack approach is easy to deploy, as it requires only brief and non-invasive access to the system (e.g., attaching a USB device takes seconds).

One way to detect the attack is that the user notices the subtle visual changes on the user interface while the attack is active (e.g., medical device settings are modified). However, our user studies show that the vast majority of users fail to notice the attack. Preventing this attack likely requires different approaches, e.g., authentication of input devices, design of protective measures on user interfaces, and this work motivates the development of such solutions. Since our attack is invisible to traditional malware detection, it operates under uncertainty without any feedback from the system, and it gives little visual indication to the user, we call it *hacking in the blind*. To summarize, we make the following contributions:

– *New attack approach.* We propose a novel way to attack systems through their user interfaces. The attack is quick to deploy, hard for users to notice, and invisible to existing malware detection.
– *New attack techniques.* We developed a novel user interface tracking and fingerprinting techniques.
– *Attack prototype.* We implemented the attack on a small embedded device.
– *User studies.* We conducted both user studies on two case UIs (online banking and implant programmer) to evaluate attack detection rates.
– *Analysis of protective measures.* We analyze possible countermeasures and point out their limitations.

2 Problem Statement

The goal of the adversary is to attack a security-critical user interface (Figure 1), and we focus on attack scenarios where the adversary has brief physical access

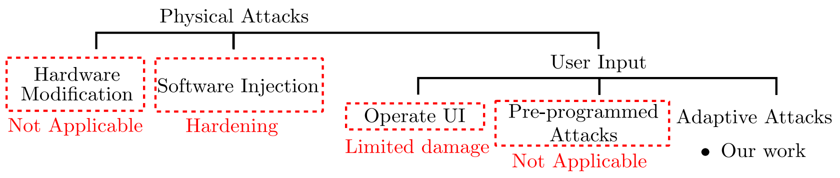


Fig. 2. Classification of physical attack techniques and their limitations to our setting.

to the target system, prior to its usage. In this section we discuss limitations of known physical and user interface attacks and describe our adversary model.

2.1 Limitations of Known Attacks

There are various kinds of physical attack, and Figure 2 provides a categorization of common attack techniques.

Hardware modification. One could argue that, in case the attacker has even brief physical access to a device, that the device should already be considered as trivially compromised. However, this is often not the case, due to two practical reasons. First, the attacker often can not shut the device down, without being noticed. This would prevent the attacker from opening the device and injecting advanced hardware backdoors or performing similar hardware modifications. Second, the attacker may simply not have sufficient time to perform such attacks. For example, in case of the hospital, we observed that the intensive care ward was never left unattended for extended periods of time.

Software injection. Another approach that relies on physical access is local injection of malicious code. For example, many terminals can be configured such that unsigned code cannot be executed from, e.g., connected USB devices.

Operate user interface. User input can not be simply blocked, and an attack approach that leverages this fact is to manipulate the device directly through its user interface. For example, if the adversary has physical access to the target device, and can operate its UI, the adversary can perform all the operations that the legitimate user is entitled to. Such attacks have two limitations. First, unauthorized use can be addressed with user authentication (e.g., devices could be screen locked). Second and more importantly, on security-critical UIs, the damage of such attacks is typically limited. For example, certain medical devices are only connected to patients when operated by doctors, and such attacks are less severe than runtime attacks that modify the operation of the terminal during its use. Similarly, e-banking sites are protected using second-factor authentication, that the attacker does not necessarily have access to.

Pre-programmed attacks. Another class of UI attacks rely on external devices connected to the target system. In such cases, no malware is present on the target system, and the purpose of the attack device is to either passively intercept user input or to actively inject pre-programmed sets of commands. A hardware key logger that collects user input is an example of a passive attack. BadUSB [19] is an example of a pre-programmed attack where a malicious input device (keyboard) injects keystrokes into the target system. Such attacks leverage keyboard shortcuts that, e.g., open a console or an administrative window and modify system settings. Such attacks do not apply to hardened terminals, as they rarely have console programs, and the user, or an adversary that controls user input, cannot “escape” the application UI to modify system settings beyond what the application enables. As the adversary does not know the current system state, such attacks can not be used to compromise (e.g., hijack) an e-banking user session, without resorting to installing malware to the device.

2.2 Our Goal: Adaptive Runtime Attacks

The focus of our work is on *adaptive runtime UI attacks*, that overcome the limitations of the above discussed techniques. We explore UI attacks that work even if hardware modifications are not practical and software injection can be prevented. Our goal is to design runtime UI attacks that are more damaging and accurate than pre-programmed attacks or operating the device through its UI.

Adversary model. We assume an adversary that gains temporary access to the target device prior to device usage, attaches a small attack device that sits in-between user input peripherals and the device (Figure 1), and leaves the premises. If the input device is external (e.g., USB mouse), the adversary can attach the attack device to the USB port that connects it to the target system. The adversary can also simply replace an external input peripheral with one that contains the attack device. Most PCs have open ports for UI peripherals, and our survey (`https://goo.gl/arp2DU`) shows that many terminals use external peripherals, also connected to easily accessible ports.

Besides installing the attack device, the adversary does not interact with the target device in any other way. In particular, we assume that the adversary cannot observe current system state (the user interface might be locked, or password protected). If the device is used via two input devices (e.g., mouse and keyboard), the adversary can connect both of them to the same attack device. The attack device can observe, delay, and block all events from the connected user input devices as well as inject new events. We assume that, in the terminal case, the adversary knows the target application UI (its states and state transitions).

Example attack targets. We explore attacks in two different contexts (general-purpose PCs and dedicated terminals) that represent different types of attack targets. The UI of a dedicated terminal device typically consists of a single application, that occupies the entire screen, and that the user cannot escape out of. Terminal devices often have fixed screen resolutions. On the other hand, general-purpose PCs have many applications with various UIs. The application UIs are managed by windowing systems that have various screen resolutions.

3 Hacking in the Blind

The installed attack device observes user events from the connected input device(s) and launches the attack by modifying user input or injecting new input events when the legitimate user is performing a security-critical operation.

While the attack device can intercept all user input events, their interpretation may have two forms of uncertainty. First, the adversary may not know the state of the target device UI (e.g., because the UI was locked when the attack device was installed). We call this *state uncertainty.* Second, the adversary may not be able to interpret all received user input events without ambiguity. In particular, mouse events are relative to the mouse cursor location that may be unknown to the adversary. We call this *location uncertainty.* In contrast to mouse input, touchscreen events do not have location uncertainty as touchscreen clicks are reported to the operating system in terms of their absolute (x, y) coordinates.

The primary challenge in our approach is to launch the attack accurately under such uncertainty, without any feedback from the target device (hacking in the blind). The best attack strategy depends on the attack platform (general-purpose PC vs. dedicated terminal), application user interface configuration, the type of the input device, and the level of stealthiness the adversary wants to achieve. We first discuss simple attack strategies, and then move on to present our two main attack techniques: state tracking and UI fingerprinting.

3.1 Simple Techniques

If the adversary manages to reduce (or remove) both location and state uncertainty, attacking the device user interface becomes easier. Assuming that the adversary knows both the current user interface state, the mouse cursor location, and complete model of the UI, each event can be interpreted unambiguously. For example, if an adversary knows the complete user interface configuration of a terminal, can easily track both mouse movement and state transitions in the user interface. In PCs, building such a model is typically not possible. Below we list simple methods that can help the adversary to reduce uncertainty.

Reducing state uncertainty. A simple technique to learn the state of the system is to wait for a reboot. If the attack device can determine when the terminal is booted, it knows that the target device UI is in a known state. While PCs are routinely restarted, many terminals run for long periods of time.

Reducing location uncertainty. A simple technique to determine the mouse cursor location is to actively move the mouse (i.e., inject movement events) towards a corner of the screen. For example, if the mouse is moved up and left sufficiently, the adversary knows with certainty that the mouse cursor is located at the top-left corner of the screen. Moving the mouse while the system is idle may not be possible, if the target device user interface is locked.

Summary. We assume a strong adversary, that in many scenarios has to perform the attack under location uncertainty, state uncertainty or both, and next we design attack techniques to handle both. We first describe state tracking that is applicable to terminals, where the adversary can build a complete model of the target system UI. After that, we describe UI fingerprinting that allows attack state detection on PC platforms, where such model creation is infeasible.

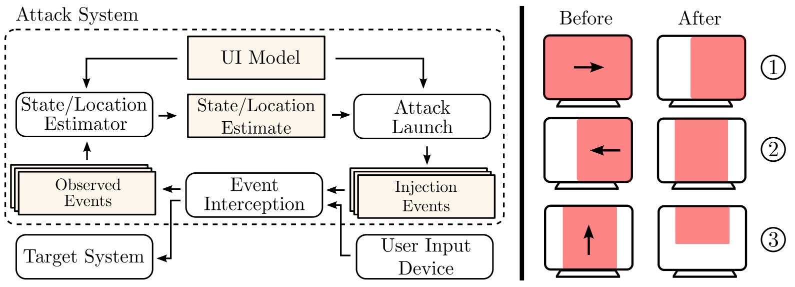


Fig. 3. (Left) Overview of our attack system. (Right) Movement event handling. Movement can reduce the uncertainty area size (1) and (3) or change its location (2).

3.2 State Tracking

Starting from this section, we describe a state tracking approach that enables the adversary to launch accurate attacks despite of uncertainty. State tracking is applicable to terminals, where the adversary can build a complete model of the target system UI (e.g., typically on dedicated terminals, the target application UI constitutes the complete target system UI). A noteworthy property of the system is that it estimates user interface state and mouse location *fully passively*, and thus enables implementation of stealthy attacks. We proceed by giving a high-level overview of the attack system (Figure 3).

The attack device contains a static model of the target system UI that the adversary constructed before the device deployment and it has two main components. The first one is a *State and Location Estimator* that determines the most likely user interface state (and mouse cursor location) based on the observed user events and the UI model. The estimation process can begin from a known, or an unknown UI state, at an arbitrary moment and it tracks mouse and keyboard events. The second component is an *Attack Launcher* that injects precise input events while the legitimate user is performing a safety-critical operation.

User interface model. The UI model contains user interface states, their elements (buttons, text boxes, sliders, etc.) and state transitions. For each state, the model includes the locations and the types of the input elements and the possible state transition that the element triggers. One of the states is defined as the start state and one or more states are defined as the target states. The goal of the attack is to modify safety-critical inputs (*target elements*) on the target states. Typically the target state includes also a confirmation element that the user clicks to confirm the safety-critical operation.

State and location estimation. Here we describe our state and location estimation algorithm for mouse and keyboard. Later we explain how the same algorithm can be used to estimate state for touchscreens (only state uncertainty).

The algorithm operates by keeping track of all possible user interface state and mouse location combinations. For each possible state and location, the algorithm maintains a *state tracker* object. The state trackers contain an identifier of the state and an *uncertainty area* that determines the possible location of the mouse in that state instance. Additionally, the algorithm assigns a probability for each tracker object that represent the likelihood that the terminal user interface and the mouse cursor are in this state and location.

The estimation algorithm maintains the tracker objects in a list. If the estimation begins from a known state, we start with only one tracker, to which we assign 100% probability. If it begins from an unknown state, we create one tracker per possible system state and assign them equal probabilities. Assuming no prior knowledge on the mouse location, we set the mouse uncertainty area to cover the entire screen in each tracker during initialization.

The state and location estimation is an event-driven process. Based on the received user input events, we create new trackers, or update and delete existing ones. For each mouse movement event, we update the mouse uncertainty area in each tracker. For every mouse click, we consider all possible outcomes of the click, including transitions to new states, as well as remaining in the same state. We create new *child trackers* with updated uncertainty areas, add the children to the list, and remove the *parent tracker*. When we observe a user event sequence that indicates interaction with a specific UI element, we update the probabilities of each tracker accordingly. We explain these steps in detail below.

Movement event handling. When the mouse uncertainty area is the entire device screen, any mouse movement reduces the size of the uncertainty area. For example, if the user moves the mouse to the right, the area becomes smaller, as the mouse cursor can no longer reside in the leftmost part of the screen (Figure 3). If the mouse is moved to a direction where the uncertainty area border is not on the edge of the screen, the mouse movement does not reduce the size of the uncertainty area, but only causes its location to be updated. Any mouse movement towards a direction where the uncertainty area is on the border of the screen, reduces the size of the uncertainty area further. For each received mouse movement event, we update the uncertainty areas in all trackers.

Click event handling. When we observe a mouse click event, the estimation algorithm considers all possible outcomes for each tracker, that are determined by the current mouse uncertainty area (Figure 4, left). For each possible outcome we create new child trackers and update their mouse uncertainty areas as follows.

If the user interface remains in the same state, the updated mouse area for the child is the original area of the parent, from which we remove the areas of the UI elements that cause transitions to other states. For each state transition, the mouse area is calculated as the intersection of the parent area and the area of the user input element that caused the transition. Once the updated mouse uncertainty areas are calculated for each child tracker, we remove the parent from the list, and add the children. We repeat the process for each tracker on the list, and we note that, as a result of this process, the list may contain multiple trackers for the same state with different mouse uncertainty areas.

The probability of a child tracker is calculated by multiplying the probability of its parent with a *transition probability*. We consider two options for assigning transition probabilities, as shown in Figure 4 (right):

- *Equal transitions.* Our first option is to consider all possible state transitions equally likely. E.g., if the mouse uncertainty area contains two buttons, each of them causing a separate state transition, and parts of the screen where a click does not cause a state transition, we assign each of them 1/3 probability.
- *Element area.* Our second option is to calculate the transition probabilities based on the surface of the user interface element covered by the mouse uncertainty area. For example, if the uncertainty area covers a larger area over one button than another, we assign it bigger transition probability.

The transition probabilities can be enhanced with *a priori probabilities* of UI element interactions. For example, based on prior experience on comparable UIs, the adversary can estimate that "OK" is pressed twice as likely as "Cancel".

Element detection. Finally, we identify user interaction with certain UI elements based on sequences of observed input events. For example, an event sequence that begins with a button down event, followed by movement left or right that exceeds a given threshold, followed by a button up event is an indication of slider usage. Similarly, text input from the keyboard indicates likely interaction with a text field, and a click indicates interaction with a button.

When we observe such event sequences (slider movement, text input, button click), we update the probabilities of the possible trackers on the list. One possible approach would be to remove all trackers from the list where interaction with the identified element is not possible (e.g., a button click is not possible under the mouse uncertainty area), and we could then increase the probabilities of the remaining trackers equally. Such an approach could yield fast results, but also provide erroneous state estimations. If the user provides text input on a user interface state that does not contain editable text fields or if text highlighting is mistaken for slider movement, the algorithm would remove the correct state from the list. We adopt a safer approach where we consider trackers with the identified elements more likely and scale up their probabilities, and keep the remaining trackers and scale down their probabilities.

Target state detection. Our algorithm continues the state tracking process until two criteria are met. First, we have identified the target state with a probability that exceeds a threshold. After each click event and detected element we sum the probabilities for all trackers that represent the same state to check if any of them exceeds the threshold. Second, the mouse uncertainty area must be small enough to launch the attack. We combine the mouse uncertainty areas from all matching trackers and consider the uncertainty area sufficiently small when its size is smaller than the size of the confirmation element.

State estimation for touchscreen. Using a touchscreen instead of a mouse does not affect our algorithm. Typically touchscreens report click events in absolute coordinates, hence using a touchscreen corresponds to the case where the mouse location is known, but the starting state is not. Determining the possible transitions after a click is trivial, since there can be at most one intersection of a clicking point with the area of an element in a specific state.

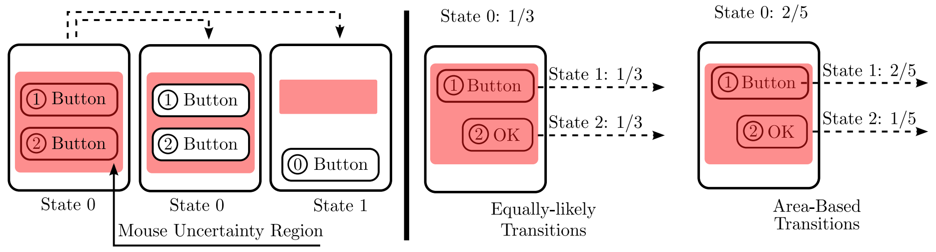


Fig. 4. Click event handling (left), and transition probabilities (right).

3.3 User Interface Fingerprinting

In this section we describe UI fingerprinting that is applicable to adaptive UI attacks on general-purpose PC platforms. On a high level, our UI fingerprinting approach works as follows. The attack device keeps a history of all events observed in the last t minutes (in our experiments, $t = 5$ minutes produced good results). For every observed mouse click event, the attack device takes the target application UI model, analyzes the event history and asks the following question: *"Is the user interacting with the critical UI?"*. Similarly to our state tracking approach, we also require a UI model for fingerprinting. However, the model is simpler as no UI transitions are modeled.

In Figure 5, we illustrate our fingerprinting approach on a concrete example. The critical UI in this case is simple, and consists of three elements: two text-boxes and a button. We require one text-box to be filled out, while the other is optional. The latest event (at time t_0) the attack device observes is a mouse click. The attack device assumes the click was performed on the confirmation element, and traverses the event history backwards in time to see if the user was indeed interacting with the critical UI, according to the specified model.

The first encountered event is a mouse move, so the device moves the mouse uncertainty region accordingly. The following two events are a click and a key press, so the device creates two trackers (the uncertainty region was over two different elements), and shrinks the uncertainty region over the corresponding elements. The next event is another move, followed by a click and a keyboard press. In the lower tracker, the click would have originated over no element, but in the upper tracker, both required text-boxes would be filled, at which point the attack device concludes that the user is interacting with the targeted UI state.

3.4 Attack Launch Techniques

Once the attack device has identified the attack state with sufficiently small uncertainty area, it is ready to launch the attack. In a simple approach, the adversary moves the mouse cursor over one of the attack elements, modifies its value, moves the cursor over the confirmation button, and clicks it. The process is fast and the user has little chances of preventing the attack. However, the user

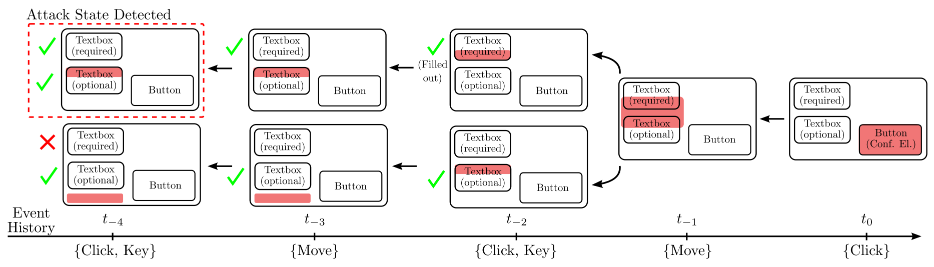


Fig. 5. Fingerprinting UI example. For every observed click, the attack device traverses the event history and checks if the user is currently interacting with the target UI.

is likely to notice it. For example, if a doctor never clicked the confirm button, the doctor is unlikely to implant the pacemaker into a patient. For this reason, we focus on more subtle attack launch techniques. Below we describe two such techniques and in Section 4 and Section 5 we evaluate their user detection.

Element-driven attack. The adversary first identifies that the user interacts with one of the target elements. This can be easily done when the mouse uncertainty area is smaller than the target element. Once the user has modified the value of the target element, the adversary waits a small period of time and during it tracks the mouse movement, then quickly moves the mouse cursor back to the target element, modifies its value, and returns the mouse cursor to its location. After that, the adversary lets the legitimate user confirm the safety-critical operation. The technique only requires little mouse movement, but the modified value remains visible to the user for a potentially long time, as the adversary does not know when the user will confirm the safety-critical operation.

Confirmation-driven attack. The adversary identifies that the system is on the attack state and lets the user to set the attack element values uninterrupted. When the user clicks the confirmation button, the attack activates. The adversary blocks the incoming click event, moves the mouse cursor over one of the attack elements, modifies its value, moves the mouse cursor back over the confirmation button, and then passes the click event to the target system. The adversary then changes the modified attack element back to its original value. In this technique, the mouse cursor may have to be moved more, but the modified attack element settings remain visible to the user only for a brief period of time.

3.5 Attack Device Protoype

We built a full prototype of the attack device by implementing the entire attack system in C++ and deployed it on two BeagleBone Black boards (Figure 6). The two boards communicate over ethernet, as each board has only one set of USB ports and we evaluate an attack where the adversary controls both mouse and keyboard input. A custom attack device would consists of a single embedded device. The boards have processing power comparable to a modern low-end smartphone (1GHz CPU, 512MB RAM).

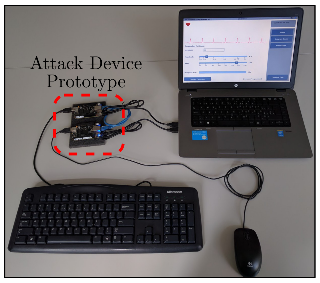
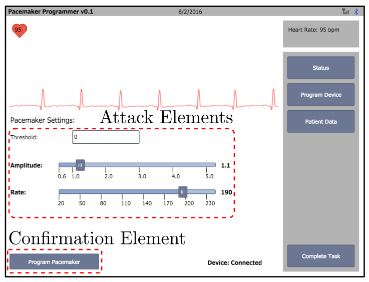


Fig. 6. Attack device prototype. We implemented the full attack.



Pacemaker Settings:

Threshold: 0

Amplitude: 1.1

Rate: 190

Device: Connected

Fig. 7. Case study UI: custom cardiac implant programmer.

The boards are conveniently powered through USB, and no external power supplies are required. Each board intercepts one USB device (keyboard and mouse, respectively), and the two boards communicate through a short ethernet cable. We emphasize that the complete attack software is running on the boards themselves, and no remote communication with the attacker is either required or performed. We purposefully optimized the C++ code for execution speed.

4 Case Study: Pacemaker Programmer UI

To evaluate our state tracking based attack on terminals, we focus on a simulated pacemaker programmer (Figure 7). We also performed a case-study on e-banking user interfaces, on 20 domain experts, and we refer the reader to Section 5 for details. A video of our attack is available at `https://goo.gl/kdkRDC`.

We implemented the user interface based on the publicly available documentation of an existing cardiac implant programmer [5]. Such a programmer terminal is used by doctors to configure medical implant settings. For example, when a doctor prepares a pacemaker for implantation, she configures its settings based on the heart condition of the receiving patient. The terminal can also be used to monitor the operation of the implant and potentially update its settings.

The model of this user interface consists of approximately ten states and contains three types of user input elements: buttons, text fields and sliders. All state transitions are triggered by button clicks. The attack elements are the user input elements that are used to configure the pacemaker settings. Threshold is set using a text field (keyboard), while amplitude and rate are set using slider elements (mouse), see Figure 7. All attack elements are on the same state. The model creation was a manual process that took a few hours.

We implemented the user interface using HTML5/JavaScript, and the UI serves two different purposes. First, we use it to demonstrate the attack and evaluate attack detection on-site. For this use, we run the user interface on a standard PC, instead of a terminal (Figure 6). Second, we use the same UI to collect user traces and evaluate the detection of different attack variants online.

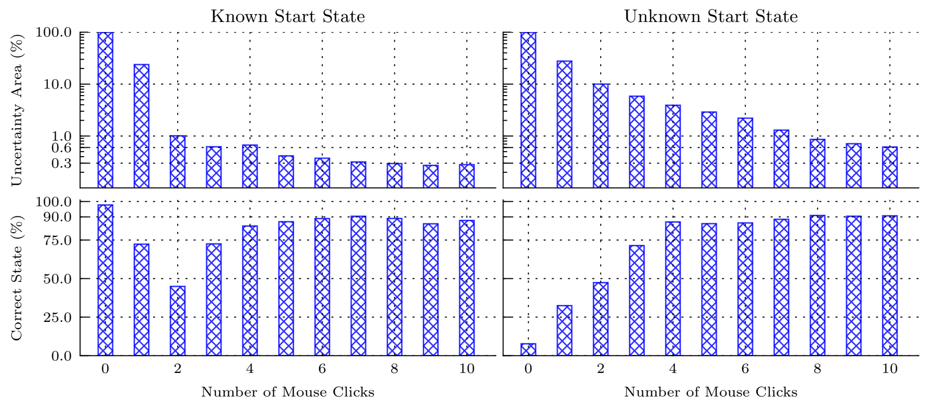


Fig. 8. State tracking accuracy of our attack system.

Trace collection. To evaluate the tracking algorithm we collected online user traces for the programmer UI. We recruited participants using the global crowd sourcing platform CrowdFlower. In the instructions, we asked the participants to find saved patient data that matches a given medical condition, copy that patient's pacemaker settings to the programming screen, and finally, to program the device by pressing the confirmation element. We recorded all user input during the task, but no private information on study participants was collected.

In total 400 contributors completed the task. We observed that approximately 7% of user input gestures were over non-existent elements (e.g., users clicked when the mouse cursor was not over a button).

Estimation accuracy. We ran our estimator on all our traces (Figure 8). As our algorithm is event-based, after each click we measured (a) the size of the mouse uncertainty area, expressed as percentage of the overall screen, and (b) the probability that we correctly estimate the real state the user is currently in.

We say that our algorithm correctly estimates the current state, when it assigns the highest probability for the correct state among all states. As all our traces start from the same state, to evaluate the situation where the tracking begins from an unknown state, we cut the first 10% from all our evaluation traces. As tracking options we used the element-area transition probabilities together with element detection (scaling parameter 0.95) and a priori probabilities that we obtained by profiling the training traces.

First, we discuss the case where the state tracking begins from a known start state (shown left in Figure 8). The uncertainty area is the full screen at first and the probability for estimating the correct state is 100% (known start state). As the estimation algorithm gathers more user input events, the uncertainty area size reduces quickly and already after three clicks the area is less than 1% of the screen size. The estimation probability decreases first, as the first click adds uncertainty to the tracking process, but after additional click events, the

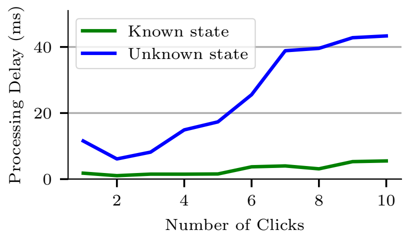


Fig. 9. The device per-event overhead increases as the algorithm accumulates more trackers, but is overall low.

Attack group	Attack succeeded	Task completed
1. Element, text, 5 ms	50%	84
2. Element, text, 62 ms	37%	84
3. Element, text, 125 ms	48%	86
4. Element, slider, 5 ms	12%	80
5. Element, slider, 62 ms	9%	83
6. Element, slider, 125 ms	6%	86
7. Confirmation, text, 10 ms	93%	81
8. Confirmation, text, 125 ms	96%	79
9. Confirmation, text, 250 ms	93%	78
10. Confirmation, slider, 10 ms	95%	85
11. Confirmation, slider, 125 ms	90%	82
12. Confirmation, slider, 250 ms	95%	79
Total		**987**

Fig. 10. Attack detection results.

estimation probability increases steadily, and after ten clicks the algorithm can estimate the correct state with 90% probability.

Next, we consider the scenario where the state tracking begins from an unknown target system state (shown right in Figure 8). In the beginning, the uncertainty area is the entire screen and the probability for the state estimate is low, as all states are equally likely. As the tracking algorithm gathers more user events, the uncertainty area reduces, but not as fast as in the case of known start state. The uncertainty area becomes less than 1% of the screen size after eight clicks. The probability for the correct state estimate increases and after ten clicks we can estimate the correct state with approximately 90% probability.

We conclude that in both cases we can identify the correct system state with high probability after observing only ten clicks and the uncertainty area becomes very small (below 1%, equals to a small, 50×50 pixel rectangle). If the user enters the attack state after ten clicks, we can launch the attack accurately.

We compared the performance of our different tracking options, and we noticed that the they performed comparably. Element detection gave a major accuracy increase, while a priori probabilities did not improve accuracy significantly.

Attack launch success rate. To evaluate if our system successfully detects the correct time to launch the attack, we ran our algorithm on all the user traces we recorded from our user study. In 83% of all traces, our system correctly identified the attack launch time, namely right after the user programs the pacemaker. In 16% of the traces, our system did not identify a suitable attack time and, as a result, no attack would be launched. Only in 1% of the traces our system launched the attack at the wrong time. We conclude that our system correctly identifies the attack launch time in most cases.

Estimation overhead. To analyze how fast our state and location estimation algorithm runs, we measured the runtime overhead of processing each user input event from the collected traces. The algorithm was run on the two BeagleBone boards with element tracking enabled, using equal transition probabilities.

Both mouse and keyboard events require little computation. The processing overhead per event is very small (below 0.5ms) and such events can be easily processed in real-time. Mouse click events require more computation, as those cause generation of new trackers, and the processing delay is relative to the number of state trackers that the algorithm maintains. Figure 9 shows the average processing delay for mouse clicks from our evaluation traces. When we start tracking from a known state, the overhead increases slightly over time, but remains under 7ms per event. When we start tracking from an unknown state, the algorithm accumulates significantly more trackers, and thus the processing overhead increases faster. After ten clicks, processing a single input event takes approximately 43ms on our test platform. From the analyzed traces we observe that the interval between consecutive mouse clicks is typically in the order of seconds. This gives the attack device ample time to process incoming click events.

We conclude that our implementation is fast. Mouse and keyboard events are processed in real-time and the processing overhead for mouse clicks is significantly smaller than the typical interval between clicks. The UI remains responsive, with no "processing lag" that would indicate an attack is taking place. Our attack device can also be significantly minimized in terms of hardware. Based on our results, even less powerful devices than our own would be sufficient.

Online attack detection user study. To evaluate how many users would detect our attacks, we conducted two user studies. Here we describe the first, large-scale online study. We recruited 1200 new study participants, that we divided into 12 groups. We tested two element-driven attack variants: one where we modify a text element and another where we modify a slider. We also tested two confirmation-driven attack variants: one with text and another with slider input. For each four attack variant we tested three separate speeds of the attack.

We provided the same task description as before, but depending on the group, we launched an attack during the task. Once the task was over, we asked the participants: *"Do you think you programmed the pacemaker correctly?"*.

If a participant noticed the UI manipulation, she had three possible ways to act on it. First, she was able to program the pacemaker again with the correct values. Second, the participant could report that the device was not programmed correctly in the post-test question. Third, the she could write to the freeform feedback that she noticed something suspicious in the application user interface.

In total 987 participants completed the task. We consider that the attack succeeded when the participant did none of the above mentioned three actions. The results are shown in Table 10. The success rate for the element-driven text attacks was 37-50% and for the element-driven slider attacks 6-12%, depending on the speed of the attack. The success rate for the confirmation-driven text attacks was 93-96% and for the confirmation-driven slider attacks 90-95%.

All the tested confirmation-driven attacks had high success rates (over 90%). In the element-driven attacks the user interface manipulation remains visible longer for the user, and this is a possible explanation why the attacks do not succeed equally well. We conclude that confirmation-driven attacks are a better strategy for the adversary and focus the rest of our analysis on those.

We compared the success rates of text and slider manipulation on confirmation-driven attacks, but found no statistically significant difference ($\chi^2(1, N = 484) =$

$0.12), p = 0.73)$. We also compared the success rates of different attacks speeds on confirmation-driven text attacks ($\chi^2(2, N = 238) = 0.42), p = 0.81$) and slider attacks ($\chi^2(2, N = 246) = 2.20), p = 0.33$), but found no significant difference. This implies that the adversary has at least a few hundred milliseconds time to perform the user interface manipulation without sacrificing its success rate.

We analyzed all freeform text responses, and none of the users associated the observed UI changes with a malicious attack. Only two users commented on the changing values of UI elements and both attributed them to a software glitch.

On-site user study. Our study participants were not domain experts, i.e., users that would commonly operate a pacemaker programmer. We therefore performed a on-site user study on two medical professionals (one doctor, one nurse). Both participants failed to detect our attack, and were under the impression they programmed the pacemaker correctly.

5 Case Study: Online Banking UI

In this section we describe our experiments on e-banking user interface. To evaluate how successful our attack is in fingerprinting UIs and compromising the integrity of e-banking payments, we performed a separate on-site user study, similar to the previous on-site attack detection study. We created partial local replicas of three major e-banking websites (we only copied the payment parts of the sites). The replicas were nearly the same as their online counterparts, with minor differences introduced through the replication process.

Recruitment and procedure. We recruited 20 domain experts, where each participant was required to be a regular e-banking user of one of the three banks we replicated the websites of, and use either Chrome of Firefox during e-banking sessions. Each participant was presented a sheet of paper with the following instructions steps. (1) Open the browser you usually use for e-banking. (2) Click on your e-banking site link, located in the browser bookmarks. (3) Imagine you already performed the login procedure, as the replica website requires no login. (4) Navigate to the payment site, and (5) make a payment to the account provided on the study sheet. (6) To complete the task, close the browser.

The attack device was already installed to the laptop and was hidden from sight. First step of our attack was to automatically detect that the user is interacting with a critical UI (payment information is filled in), and which of the three banks was being used. The second step was to detect when the "Confirm payment" button was clicked. The attack device then inserted mouse and keyboard input that changed the payment field containing the amount of money transmitted, and injected a javascript snippet through the URL bar that masked the changed value in the upcoming "Payment confirmation" screen. The javascript was only needed to mask the website so that the user has a harder time noticing the attack. The whole attack was done in approximately 0.5 seconds, and a video which demonstrates our attack is available at: `https://goo.gl/kdkRDC`. After completing the user study, we presented each participant with an exit questionnaire: *"1. Was the payment experience comparable to your regular e-banking experience?"* and *"2. What do you think the user study was about?"*

Results. Our attack successfully detected the precise point in time when the users were interacting with the critical UI (making a payment) in 90% of the cases. Our attack failed in only 10% of the cases (two users). In both cases the attack state detection succeeded, but the attack input injection failed due to implementation flaws, that were corrected after the study.

90% of participants positively answered to the first question, noting that it was similar to their regular e-banking experience. Out of those, some users noted that the UI looked slightly different, which was due to the imperfections introduced by our replication process. Only 10% noted that the experience was not the same, due to the missing second-factor authentication step. Out of 20 participants, 30% had no idea about the true nature of the user study. Another 30% suspected that some form of attack was performed (phishing, key-logging, removal of second-factor authentication), while another 30% thought the study was a usability test. Only two users detected our attack, and correctly guessed the true nature of the study. We conclude that our attack was stealthy.

Discussion. We did not perform any security priming in our user study, however we acknowledge that the role-playing bias of study participant not using their real e-banking could be present. Users might have been less careful, because they knew that their own money is not at risk. At the same time, our study setup introduces another bias. Since the study was performed under our supervision, some study participants may have been more alert than if they would have done the online payment on their own.

We tested our attack on various browsers, e-banking sites and screen resolutions as well as browser window locations, and we conclude that our attacks can successfully perform UI fingerprinting and e-banking session hijacking, with no false positives, and very low false negatives. Furthermore, we showed that our attacks were not detected by the majority of users.

The user study was performed on our custom laptop, and the e-banking website were replicas. At no point did we require the study participants to disclose any kind of private information, such as their their e-banking credentials.

6 Countermeasures

In this section we analyze possible countermeasures and their limitations.

Trusted input devices. One way to address our attacks is to mandate usage of trusted input devices. We call a user input device *trusted*, when it securely shares a key with the target system, as then all traffic can be encrypted and authenticated which prevents the adversary from observing, injecting, or replaying events. However, deployment of secure input devices is challenging. For example, doctors need be able to operate medical terminals at all times, even in case of trusted peripherals breaking down.

Increased user feedback. The UI can provide visual feedback on each change of attack elements [12]. For example, the UI can draw a border around a recently edited element to keep it visible. In a confirmation-driven attack, the user would see the border, but as the adversary changes the attack element value back to the original, the content of the user interface element would appear as expected. Noticing that an attack happened may be difficult for the user.

Change rate limiting. The user interface could limit the rate at which the values of the UI elements can be changed. However, our study results show that the majority of users do not notice even relatively slow UI manipulations that take 250ms. Finding a rate limit that efficiently prevents user interface manipulation attacks, but does not prevent legitimate user interactions is challenging.

Randomized user interfaces. Another way to address our attacks is to randomize parts of the safety-critical system user interface. While UI randomization [23,26,13] can complicate, or even prevent our attacks, it also increases the chances of human error. In contrast to smartphone screen lock, on safety-critical terminals an increased error rate is typically not acceptable. For example, medical devices consider lack of UI consistency a critical safety violation [10].

Continuous user authentication. While traditional user authentication systems require the user to log in once, continuous authentication systems monitor user input over a period of time to detect if the usage deviates from a previously recorded user profile. Many such systems track mouse velocity, acceleration and movement direction [22,6], together with click events [22,6], angle-based curvature metrics and click-pause measurements [28].

The proposed systems [28,17,21] need to observe the impostor for significant amount of time (e.g., 12 consecutive seconds [21]). Our attacks require only brief mouse movement and one or few clicks, and the attacks can be performed well under a second. Our state estimation works fully passively, and current continuous authentication systems are not directly applicable to detecting them.

Summary. We conclude that all the reviewed countermeasures have limitations. Finding better protective measures that are both effective and practical to deploy remains an open problem.

7 Discussion

In this section we discuss the applicability of the proposed approach to other scenarios and directions for future work.

Attacks in the wild. Attacks similar to ours may already be taking place in the wild. For example, the NSA cottonmouth project [1] is a malicious USB connector that can both inject and observe user input. Such and similar devices are ideally suited to perform our attacks.

User interface complexity. We experimented our attacks on a terminal user interface that consists of approximately ten states. We consider this typical UI complexity for embedded dedicated-purpose terminals. We also experimented our attack on real (and replica) online banking websites. We believe that these examples capture different types of security-critical user interfaces.

System output. In our attack scenario, the adversary has only access to the user input channel. An attack device that is attached to an interface that connects a touchscreen to the terminal mainboard is an example of a scenario where the adversary may be able to access the system output channel as well. Video interfaces can have high bandwidths and running image recognition algorithms on a small embedded device in real-time may be challenging.

User presence. We tailored our attack for the case, where the legitimate user is operating the device. The presence of the legitimate user both helps and complicates our attack. Observing specific input events (e.g., mouse clicks that presumably take place over buttons) help the adversary to determine the current user interface state. At the same time, the adversary must inject the attack events in a subtle manner to avoid user detection. If the attack is performed without the user presence (i.e., when the system is idle), a different strategy is needed. Exploring such state estimation strategies is an interesting future work.

Mouse transfer function. In our attack prototype, we assumed there was no mouse transfer function, and that the physical mouse movement directly corresponded to the on-screen cursor movement. However, if present, such functions can be inferred and accounted for [20].

8 Related Work

USB attacks. Key loggers are small devices that the adversary can attach between a keyboard and the target system. The key logger records user input and the adversary collects the device later to learn any entered user secrets such as passwords. Such attacks are limited to passive information leakage, while our approach enables active runtime attacks with severe safety implications.

A malicious user input device, or a smartphone that impersonates one [27], can attack PC platforms by executing pre-programmed attack sequences [7,9,15]. For example, a malicious keyboard can issue dedicated key sequence to open a terminal and execute malicious system commands. The input device might also be able to copy malicious code to the target system. Such attacks are typically not possible on hardened embedded terminals where the user cannot escape the application UI, and installation and execution of unsigned code is prevented.

An attack that sounds similar to our attack approach is Mousejacking [2]. However, in the Mousejack attack, the attacker does gain access to legitimate user input, but is only able to blindly inject fake events into the input channel.

USB firewalls. In recent research, USB firewall architectures have been proposed [24,3,25]. Similar to network firewalls, these approaches include packet filtering functionality (e.g., in the OS kernel), and can prevent a USB peripheral of one class masquerading as an instance of another class (e.g., mass storage device masquerades as keyboard). Such measures do not prevent our attacks, where all injected USB packets match the device class of the benign peripheral.

USB fingerprinting. Researchers have demonstrated fingerprinting of PCs based on their USB communication timing patterns [4]. Similar approach could be applied to fingerprint USB input devices. The processing delays that our attack incur are so small that users cannot observe them, but it remains an open question if timing-based fingerprinting could be used to detect the attack.

Terminal protection. Software-based attestation is a technique that allows a host platform to verify the software configuration of a connected peripheral [14]. Such attestation would not address the variant of our attack where the attack device sits between the benign peripheral and the terminal. Power analysis can be used to identify unknown (malicious) software processes running on embedded terminals, such as medical devices [8]. Such approaches would not detect our attack where no malicious code is running on the terminal. Our prototype is susceptible to power-analysis as it draws power from the host USB connection. However, the device could easily be designed to include an on-board battery.

User interface attacks. In systems where multiple applications or websites share the same display, the user can be tricked to interact with false UI elements. For example, a malicious website may be able to draw an overlay over a button that causes the user click the button unintentionally. Such attacks are called clickjacking [12] or UI redressing [18]. In our attack scenario, the adversary can only modify and injects user events.

9 Conclusions

In this paper we have presented hacking in the blind, a novel approach to attack systems through input integrity violation under uncertainty about the target system state. In the attack, the adversary installs an attack device between a user input device and the target system, and the attack is launched when the authorized user is performing a security-critical operation, by modifying or injecting new user input events. Our approach is easy to deploy on the location, invisible to traditional malware detection, difficult for the user to notice, and surprisingly robust to noise. Many of the attack variants we tested had success rate over 90%. We analyzed several countermeasures and noticed that all of them have limitations. We conclude that our attack presents a serious threat to many safety-critical terminals and PC applications.

References

1. Nsa cottonmouth project. `https://nsa.gov1.info/dni/nsa-ant-catalog/usb/index.html`.
2. Mousejack technical details, 2017. `https://www.bastille.net/research/vulnerabilities/mousejack/technical-details`.
3. S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals, 2015. `http://arxiv.org/abs/1506.01449`.
4. A. M. Bates, R. Leonard, H. Pruse, D. Lowd, and K. R. Butler. Leveraging usb to establish host identity using commodity devices. In *Network and Distributed System Security Symposium (NDSS)*, 2014.
5. Biotronik. Cardiac rhythm management. `http://goo.gl/jvCuzC`.
6. Z. Cai, C. Shen, and X. Guan. Mitigating behavioral variability for mouse dynamics: A dimensionality-reduction-based approach. *Transactions Human-Machine Systems*, 44(2), 2014.

7. K. Chen. Reversing and exploiting an apple firmware update. In *Black Hat USA*, 2009.
8. S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *USENIX Workshop on Health Information Technologies (HealthTech)*, 2013.
9. A. Crenshaw. Plug and prey: Malicious usb devices, 2011.
10. M. Graham, T. Kubose, D. Jordan, J. Zhang, T. R. Johnson, and V. L. Patel. Heuristic evaluation of infusion pumps: implications for patient safety in intensive care units. *Journal of Medical Informatics*, 2004.
11. Hak5. Usb rubber ducky, 2017. `http://usbrubberducky.com/`.
12. L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, 2012.
13. Intel. Identity protection technology with pki - technology overview, 2013. `https://goo.gl/TtgzXW`.
14. Y. Li, J. M. McCune, and A. Perrig. Sbap: Software-based attestation for peripherals. In *Trust and Trustworthy Computing (TRUST)*, 2010.
15. J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham. Mouse trap: exploiting firmware updates in usb peripherals. In *Workshop on Offensive Technologies (WOOT)*, 2014.
16. Microsoft. Lockdown features (windows embedded industry 8.1), 2014. `https://goo.gl/JcXC9X`.
17. S. Mondal and P. Bours. A computational approach to the continuous authentication biometric system. *Information Sciences*, 304(C), 2015.
18. M. Niemietz. Ui redressing: Attacks and countermeasures revisited. In *CONFidence*, 2011.
19. K. Nohl and J. Lell. Badusb – on accessories that turn evil. In *Black Hat USA*, 2014.
20. P. Quinn, A. Cockburn, G. Casiez, N. Roussel, and C. Gutwin. Exposing and understanding scrolling transfer functions. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 341–350. ACM, 2012.
21. C. Shen, Z. Cai, X. Guan, Y. Du, and R. Maxion. User authentication through mouse dynamics. *Information Forensics and Security, IEEE Transactions on*, 8(1), 2013.
22. C. Shen, Z. Cai, X. Guan, H. Sha, and J. Du. Feature analysis of mouse dynamics in identity authentication and monitoring. In *IEEE International Conference on Communications (ICC)*, 2009.
23. G. P. Store. Lockdown pro. `https://play.google.com/store/apps/details?id=appplus.mobi.lockdownpro`.
24. D. J. Tian, A. Bates, and K. Butler. Defending against malicious usb firmware with goodusb. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
25. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making usb great again with usbfilter. In *To appear in USENIX Security Symposium*, 2016.
26. E. von Zezschwitz, A. Koslow, A. De Luca, and H. Hussmann. Making graphic-based authentication secure against smudge attacks. In *International Conference on Intelligent User Interfaces (IUI)*, 2013.
27. Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.
28. N. Zheng, A. Paloski, and H. Wang. An efficient user verification system via mouse movements. In *Computer and Communications Security (CCS)*, 2011.