Universiteit Leiden
The Netherlands

Opleiding Informatica

VeryBadUSB:

Expanding the Capabilities of BadUSB with USB Type-C

A.J. van den Heuvel

Supervisors:
Olga Gadyatskaya & Jafar Akhoundali

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 15, 2025

Abstract

BadUSB attacks exploit the implicit trust that computers grant to USB devices, allowing malicious peripherals to impersonate keyboards or other trusted devices and execute unauthorized actions. VeryBadUSB is an evolution of this attack paradigm that leverages modern USB Type-C features and real-time computer vision to enable stealthier and more precise exploits.

In this thesis, we design and implement a proof-of-concept USB-C attack tool that can autonomously recognize on-screen elements, specifically the user's mouse cursor and a chosen clickable target, using a YOLO-based object detection model running on the device. To train this model without relying on labor-intensive real screenshots, we developed a custom synthetic data generation pipeline that produces annotated images of a standard Ubuntu 22.04 desktop with various cursor and icon positions. The resulting YOLO model, trained entirely on synthetic data, successfully detects the actual cursor and target icon in live screen captures of the target system. This enables the device to locate UI elements in real time as a prelude to context-aware malicious actions, all without immediate command injection that might trigger security alarms. Our experimental evaluation confirms the feasibility of this approach: the vision-enhanced device reliably identifies GUI targets and can interact with them under controlled conditions. We document a reproducible setup for the entire system, from data generation to model deployment, to encourage further research. The outcomes demonstrate that incorporating visual reconnaissance into BadUSB-style attacks is practical and enhances an attacker's situational awareness.

In summary, VeryBadUSB extends BadUSB's capabilities by using USB-C's multi-modal I/O and onboard AI to perform initial stealthy reconnaissance, laying the groundwork for more adaptive exploitation on modern systems. This thesis aims to raise awareness about the seriousness of this attack and alert defenders to prioritise the protection of USB-equipped systems.

Contents

1 Introduction & Motivation

Cybersecurity remains a concern as organizations and individuals face more and more cyberattacks. While remote attacks executed over the Internet often dominate defensive attention, local attacks via physical devices are an equally important threat vector. In particular, USB devices plugged directly into a system can bypass many traditional security layers. Most operating systems implicitly trust USB devices by default, granting them the same privileges as standard peripherals. This implicit trust makes USB interfaces a powerful tool for stealthy exploits. BadUSB-style attacks are a prime example: a malicious USB device can be reprogrammed to impersonate a keyboard or mouse, quietly injecting unauthorized commands into the host without user consent. Such attacks illustrate the danger of trust-by-default. However, traditional BadUSB payloads operate blindly, relying on fixed scripts with no awareness of the host's on-screen state or context.
This thesis is about the next generation of BadUSB that uses modern USB Type-C capabilities to its advantage, given that context-aware attacks have a higher success rate. USB Type-C supports display output alongside input devices. That is why in this thesis we design a way to exploit this specification, by analyzing the victims screen. A real-time object detection model (YOLO) is used to detect the victims cursor and potential targets. To train this model we developed a lightweight synthetic dataset for Ubuntu 22.04. In experiments, the trained model succesfully recognizes the actual on-screen cursor and target icon about 15 times per second. This capability allows the malicious device to perform initial reconnaissance, ultimately making following malicious actions more precise and stealthier.

This leads us to the central research question: *To what extent can a USB Type-C device with onboard computer vision perform real-time reconnaissance by locating the mouse cursor and UI targets on a typical Linux desktop?*

By combining computer vision with USB-based access, we demonstrate how an attacker could integrate a visual reconnaissance phase into a BadUSB attack. Gaining this on-screen awareness beforehand enables more context-aware exploitation of the target system, which might result in a stealthier attack.

The research was conducted at the Leiden Institute of Advanced Computer Science (LIACS) under the supervision of Jafar Akhoundali and Olga Gadyatskaya. Their supervision was key to the outcome of this thesis, and their guidance, feedback, and encouragement throughout the process are gratefully acknowledged.

1.1 Thesis Overview

This thesis is structured as follows: Section 2 introduces the threat model, system assumptions and ethical considerations. Section 3 discusses related work in the area of USB-based attacks and vision-based exploits. Section 4 presents the methodology, showing the synthetic dataset generation pipeline and annotation process. Section 5 explains the YOLO model training and optimization steps. Section 6 reports the experimental setup and evaluation results. Section 7 discusses limitations and implications, and finally, Section 8 concludes the thesis and outlines future research directions.

2 Threat Model & Ethical Considerations

We consider a local USB-based attack scenario in which an unsuspecting victim connects a malicious USB Type-C device to their computer. The device is planted or presented by an attacker but masquerades as a charging cable or a USB hub/dock so the victim has no reason for suspicion. Upon connection, the rogue hardware exploits the host's trust-by-default behavior: it simultaneously enumerates as a standard Human Interface Device (HID) (e.g. a keyboard/mouse) and as a video interface (e.g. a USB display adapter or webcam). Through this dual impersonation, the device can inject input events (keystrokes or mouse movements) into the system while also capturing screen output in real time. Crucially, the attacker's device operates with no special privileges or malware on the host. It abuses only the legitimate capabilities granted to USB peripherals. The operating system automatically recognizes the device's declared functions and enables them without user authorization, allowing the attacker to perform stealthy on-screen reconnaissance.

In our implementation, the target machine is an Ubuntu 22.04 LTS system with a default GNOME desktop environment, running at a 1920×1080 screen resolution. We assume the machine has out-of-the-box settings and no special USB hardening (such as device white-listing or user prompts) in place. Once connected to this system, the malicious USB-C device's immediate goal is GUI reconnaissance rather than active exploitation. It leverages an onboard computer vision module (a YOLO-based object detector) to analyze the live screen feed and identify specific interface elements. In particular, our tool locates the user's mouse cursor and a chosen target icon on the desktop. By monitoring the screen content, the device gains contextual awareness of the user's session without triggering side effects. We intentionally enclose the attack to this reconnaissance phase. No system commands are executed and no persistent changes are made during this stage. This conservative approach keeps the attack covert, avoiding tell-tale signs (like sudden cursor movements or unauthorized windows) that might alert the user or security software. The captured GUI information could theoretically be used to plan a follow-up action (e.g. precisely clicking the target icon at an opportune moment), but those steps lie outside the scope of our study's initial phase.

2.1 Ethical Considerations

This research was conducted under white-hat principles. All experiments were performed on hardware and computers owned by the researchers, ensuring that no unsuspecting users or external systems were put at risk at any time. Notably, we did not develop a deployable BadUSB attack tool for malicious use. Our product is a proof-of-concept prototype used only to demonstrate potential vulnerabilities. The aim is to raise awareness of USB threats among the security community. We disclose our findings responsibly, with the intent of strengthening system protections. In summary, this thesis explores what could be done by an attacker in theory, but it does so to help practitioners recognize and mitigate such threats. No part of the project was used offensively.

3 Related Work

3.1 Early USB Threats and Emergence of BadUSB (2000s–2014)

USB devices have long posed security risks as vectors for malware and social engineering. Early attacks often involved USB flash drives as infection media, from the 2008 Conficker worm exploiting Windows AutoRun, to the 2010 Stuxnet attack propagating via USB drives to sabotage Iranian nuclear facilities [DA24] [Gre14b]. Users have also been susceptible to plugging in unknown USB sticks, as demonstrated by Stasiukonis (2006) leaving malware-laden drives in parking lots to social-engineer victims [SGRY17] [TSB+16]. These incidents underscored the USB ecosystem's "trust-by-default" assumption [TSK+18]: when a device is plugged in, hosts trust it and enable its functions without authentication. This trust opened the door for more advanced USB-borne attacks.

A shift occurred in 2014 with the public disclosure of BadUSB, a new class of USB firmware attacks. In a Black Hat 2014 briefing, Nohl and Lell revealed that a USB device's firmware (e.g. on a flash drive's controller) can be reprogrammed to masquerade as a different device type, such as a keyboard, and issue malicious commands on the host in the background [GPS16]. By emulating a trusted Human Interface Device (HID) like a keyboard or mouse at the firmware level, a "BadUSB" device could invisibly take control of a computer, typing commands to download malware or open backdoors without user consent [GPS16] [Bla18]. Notably, BadUSB attacks reside in device firmware (outside the host's file system), making them invisible to traditional antivirus scanning and surviving reformatting of the device [Bla18]. The original demonstration showed several ominous capabilities: for example, an infected USB drive spoofed a network adapter to reroute the victim's web traffic to attacker-controlled servers, and an Android phone (when plugged in) impersonated a USB keyboard to bypass two-factor authentication [Bla18]. These proofs-of-concept highlighted that 'once infected, computers and their USB peripherals can never be trusted again,' unless USB firmware security is fundamentally redesigned [Bla18].

Within months of the 2014 revelation, other researchers released proof-of-concept code for BadUSB-style exploits, making the threat widely accessible [NKL14] [Gre14a]. Security enthusiasts demonstrated DIY BadUSB devices using microcontroller boards. For instance, Samy Kamkar used a Teensy microcontroller to create a "USB drive-by" device that, when connected to an unlocked PC, emulated a keyboard to change DNS settings and install a backdoor. Likewise, Mittal's Kautilya toolkit provided a library of HID payload scripts for keystroke injection attacks (e.g. launching a reverse shell). By 2015, penetration testers were routinely using malicious USB dongles (such as the "USB Rubber Ducky") to evaluate corporate security awareness, leveraging the fact that a USB interface disguised as a keyboard is inherently trusted on most systems [VVR15] [GPS16]. Vouteva's 2015 study on BadUSB feasibility systematically implemented such attacks using an Arduino Micro programmed as a USB keyboard [VVR15]. Her results confirmed not only that HID-emulation attacks are technically feasible, but also that they can succeed in seconds and yield full remote control of a target without needing pre-installed malware [VVR15]. However, early exploits had practical limitations – many off-the-shelf payloads required the victim to have administrator privileges or to approve User Account Control (UAC) prompts, which could hinder the attack in standard user environments [VVR15]. This drove attackers to craft more stealthy

techniques (e.g. leveraging built-in system tools and PowerShell scripts) that could execute under user-level accounts while evading UAC [VVR15].

In parallel, the USB Implementers Forum (USB-IF) and security community began acknowledging these new threats. USB-IF's initial recommendation was to secure devices by enforcing signed firmware – i.e. only allow firmware updates with vendor signatures [GPS16]. While firmware authentication can harden devices against becoming BadUSB, it does not protect hosts that still trust any plugged-in device by default [GPS16]. Recognizing this gap, researchers started exploring host-side defenses and usage guidelines to mitigate BadUSB.

3.2 Defenses: Host Protections and Firmware Scanning (2015–2017)

The first wave of BadUSB defenses aimed to detect or restrict unexpected device behavior on the host. GoodUSB (Tian et al., 2015) [TBB15] proposed a host-based policy framework to counter malicious device reprogramming [GPS16]. In GoodUSB's approach, when a new USB device is inserted, the user is prompted via a GUI to explicitly authorize the device's intended functionality (e.g. "mass storage only" or "keyboard input") [GPS16]. The host then blocks any device capabilities beyond the user's stated expectation, thwarting devices that try to spoof additional interfaces. This "distrust-by-default" model – essentially whitelisting device functions – addresses the root cause of BadUSB by revoking the USB stack's implicit trust [LWL+21]. However, GoodUSB and similar software solutions assume a vigilant user and a compliant operating system. A careless user could still approve a fraudulent device, and a compromised OS might be unable to enforce USB policies [GPS16].

Other host-side defenses emerged to automatically detect anomalous USB activity. USBFILTER (Tian et al., 2016) [TSB+16] intercepts USB device communications at the OS level to filter suspicious commands or enforce class-based policies (e.g. disallow a device that enumerated as a storage drive from suddenly acting as a keyboard) [TBB15][TBBR16]. Similarly, USBGuard (an open-source Linux tool) and research prototypes like USBlock (Neuner et al., 2018) block unauthorized HID keystroke injection by monitoring the context and timing of input events [NVF+18]. On another front, researchers focused on the devices themselves: Hernandez et al. (2017) developed FirmUSB, a framework for analyzing USB device firmware images via symbolic execution to identify firmware that covertly contains BadUSB logic [HFT+17]. Such firmware vetting tools can help vendors and investigators detect compromised USB peripherals before they are used in the wild.

A notable hardware-based defense is USBCheckIn by Griscioli et al. (2016) [GPS16]. USBCheckIn is an external USB "firewall" that interposes between any USB device and the host, and it forces a physical human verification for devices claiming to be HIDs. The idea is elegant: when a keyboard or mouse is plugged in, USBCheckIn will challenge the user to perform a simple task (e.g. type a displayed random code or move the mouse in a specified pattern) before allowing input to reach the host [GPS16]. A genuine human-interface device in the user's hand can easily comply, but a malicious impostor (BadUSB device with no actual keyboard hardware for the user to type on) cannot produce the correct response [GPS16]. This approach effectively authenticates the user's presence along with the device. Because USBCheckIn is a dedicated hardware proxy, it can operate even during boot or in "air-gapped" scenarios, and it does not rely on the host's OS security

[GPS16]. The downside, however, is the added cost and inconvenience of an inline device for every USB port [LWL+21]. In practice, such solutions may be adopted in high-security environments, whereas general users and organizations have been slow to deploy additional USB hardware checks.

Defenses in this era were somewhat fragmented – each tackling a subset of the problem (firmware tampering, rogue HID input, etc.) – and not yet widely built into commercial platforms [TSK+18]. A 2018 systematization of USB insecurity (Tian et al.) found that most defenses focus on one layer of the USB communication stack, while real attacks often span multiple layers (physical, protocol, OS, and human factors) [TSK+18]. For example, restricting HID inputs does nothing against a USB Ethernet device that launches a network-based attack, and vice versa. This led to a consensus that a more holistic, "multi-layer" defense strategy is needed. Indeed, the USB-IF's Type-C Authentication specification (released in 2016) was an attempt at a holistic fix: it defines a cryptographic handshake to authenticate USB Type-C devices (e.g. to ensure a device is certified and not tampered) before the host fully trusts it [TSK+18]. In theory, this could prevent untrusted or cloned devices from even interacting with the host's USB stack. However, Tian et al. (2018) formally analyzed the Type-C Authentication protocol and uncovered design flaws that undermine its security guarantees [TSK+18]. For instance, an attacker could exploit gaps in the spec to bypass authentication or perform attacks not covered by the verification scope [TSK+18]. The researchers noted that had the Type-C designers studied prior USB attack patterns, many of these flaws could have been avoided [TSK+18]. As of 2021, Type-C authentication was not widely implemented in consumer devices, and the USB ecosystem remained largely vulnerable to BadUSB-style exploits on legacy and modern ports alike [TSK+18].

3.3 Expanded Attack Surfaces with USB 3.x and Type-C (2016–2019)

While defenders worked on patches and policies, attackers continued to evolve BadUSB techniques, especially by leveraging the new USB 3.0/3.1 Type-C connectors that rose to prominence after 2015. USB Type-C introduced a physically symmetrical port with support for higher data rates and alternative protocols (DisplayPort, HDMI, Thunderbolt, etc.), consolidating many functionalities into one interface. By 2016–2017, Type-C ports had become the universal I/O on flagship laptops and smartphones. Unfortunately, the increased capability of Type-C came with an expanded attack surface. Research showed that "USB innovation has largely left security as an afterthought", repeating past mistakes in the rush to adopt new features.

One set of threats exploits the USB protocol and drivers more directly. For instance, security researchers demonstrated USB fuzzing and driver exploits – sending malformed descriptors or rapid re-enumerations to crash or hijack host systems [Sch14] [XSL+24]. Schumilo et al. (2014) showed how to find vulnerabilities in USB device driver stacks via fuzz-testing, uncovering bugs that could lead to kernel compromise when a malicious device is plugged in [Sch14] [BG12]. Other work by Su et al. (2017) presented a USB signal interference attack that can sniff keystrokes from an adjacent USB port by measuring electromagnetic leakage ("USB Snooping") [SGRY17] [Sta06]. These low-level attacks target the USB physical and link layers, proving that BadUSB is not the only peril – even a device that doesn't pretend to be a keyboard might exploit electrical/driver behavior to undermine a system.

Another emerging vector was power and alternate-mode abuse. The USB Type-C cable supports high-wattage charging and video output, which led to attacks like the "USB Killer" (a malicious thumbdrive that rapidly discharges high voltage into the host to physically destroy hardware) and faulty Type-C cables that could electrically damage devices by shorting pins. More pertinent to cyber-attacks is the Thunderbolt-over-USB-C functionality. Type-C ports often integrate Thunderbolt 3, which extends the PCI Express bus to external devices. This enabled DMA attacks: a malicious Thunderbolt peripheral (or adapter) can directly read/write system memory, bypassing CPU oversight. The 2019 Thunderclap study demonstrated how a compromised Thunderbolt device could break sandboxing and execute code with kernel privileges on Windows, Linux, and macOS, despite IOMMU protections [War19] [Gat19]. In effect, plugging in a rogue Type-C dock or adapter might allow an attacker to plant malware via DMA – an attack quite distinct from BadUSB's HID emulation, but facilitated by the same "all-in-one" USB-C port. These hardware-level attacks reinforced the notion that any unrestrained direct access (whether as a HID, storage device, or Thunderbolt peripheral) can be a weak link.

By the late 2010s, academic attention had produced a comprehensive view of USB insecurity. Nissim et al. (2017) catalogued 29 USB-based attacks, ranging from autorun malware and keystroke injection to electrical sabotage and side-channel leaks [NYE17]. They categorized attacks into four groups: those targeting the host via malicious devices, those targeting devices via malicious hosts, those leveraging user deception (social engineering), and those exploiting electrical/physical layers. This taxonomy highlighted that BadUSB (firmware reprogramming to impersonate devices) was just one category — albeit one of the most impactful because it bridges human trust and technical exploit. The arms race continued into practical use: by 2020, real-world adversaries were actively employing BadUSB tactics. The FBI reported that FIN7, a notorious cybercrime group, mailed out malicious USB sticks to U.S. companies (sometimes packaged with gift cards or teddy bears as lures) to deliver ransomware [Wik24]. When plugged in by unsuspecting employees, these devices emulated keyboards that launched PowerShell scripts, infecting the PCs with REvil/BlackMatter malware [Wik24]. A follow-up FBI alert in 2022 confirmed that FIN7 expanded this USB mail campaign to defense and transportation sectors, impersonating Amazon and government agencies in the phony letters [Wik24]. These incidents underscore that the BadUSB attack, once a theoretical demo, has matured into a real threat vector for initial access in cyber-attacks, the very scenario of interest for this thesis.

3.4 Multi-Modal Attacks and Visual Side-Channels (2018–2021)

As attackers sought to increase the potency and stealth of BadUSB, some began combining USB attacks with other modalities such as video and network exploits. One notable trend is the use of "multi-interface" USB devices that present more than one persona to the host. For example, a single Type-C device can enumerate as a keyboard, a network card, and a storage drive simultaneously (or in sequence after re-enumeration) [GPS16]. This versatility enables complex attack chains: the fake network adapter can hijack DNS or network traffic (as shown by Nohl/Lell and Kamkar in 2014), the fake keyboard can inject commands, and the mass storage can auto-run a payload or exploit the host's file parser. The Bash Bunny (a penetration testing tool released in 2017) exemplifies this approach. The Bash Bunny is a USB stick that can act as multiple devices and carry out scripted multi-stage attacks (e.g. become an Ethernet interface to bypass network restrictions, then switch

to HID to exfiltrate data).

A more recent branch of research has explored visual side-channels in conjunction with USB. The idea is that a malicious USB device might not only send keystrokes, but also capture video or images from the host or environment to gather intelligence and adapt its payload. Early signs of this appeared in the "Juice Filming" attack proposed around 2017. In a Juice Filming Charging (JFC) attack, a charging station or cable secretly records the screen of a smartphone while it charges [Res]. Meng et al. (2018) showed that through the Mobile High-Definition Link (MHL) feature (available on some micro-USB/Type-C ports), a malicious charger can automatically tap into a phone's display output without any user permission, spying on everything the user sees and types during charging [Res]. The attacker simply implants a tiny HDMI capture device inside a charger or cable. This multi-modal attack combines a power supply (which users inherently trust) with a video tap, illustrating the expanding scope of USB-based threats. The JFC attack could harvest passwords, messages, or private photos displayed on the phone, all while the user believes they are only charging their device. It required no malware on the phone – the USB connection itself carried the video feed out to the attacker [Res]. While JFC specifically targeted smartphones, the concept of a USB device capturing a host's screen or camera can apply to PCs as well (e.g. a compromised USB-C monitor cable or docking station that records screen content).

The pinnacle of multi-modal attack research is BadUSB-C, a model introduced by Lu et al. in 2021 that directly inspires my work. BadUSB-C revisits the original BadUSB concept with the new USB Type-C features in mind [LWL+21, Section I]. The authors observed that traditional BadUSB attacks have "several limitations". Notably, the attacker lacks visual feedback and UI context, making the attack less precise [LWL+21, Section I]. Classic BadUSB payloads often blindly launch a command prompt and execute scripts, but cannot react dynamically to what's happening on the screen (e.g. waiting for the user to log in or finding a specific on-screen button) [LWL+21, Section I]. As a result, they tend to stick to text-based interactions, which can be limiting or detectable (for instance, a command-line download might be flagged by antivirus if it triggers network use [LWL+21, Section I]). BadUSB-C addresses this by exploiting USB-C's ability to carry alternate data modes, specifically video. Many USB-C ports support standards like DisplayPort Alternate Mode or UVC (USB Video Class) for webcams. BadUSB-C takes advantage of this by operating in multiple modes: it can act as a HID keyboard/mouse and simultaneously as a video capture device [LWL+21, Section I]. In their implementation, the malicious device first invisibly mirrors the victim's screen through the USB-C connection (e.g. enumerating as an external display or camera) to obtain the graphical UI state, then uses that information to precisely time and target its injected HID keystrokes [LWL+21, Section I]. This is a game-changer, because the attack is no longer blind. For example, BadUSB-C can detect when a sensitive app or website is open and then execute context-specific actions. Lu et al. demonstrated three modes: (1) HID Injection Mode, which delivers keystroke payloads (for installing malware or creating backdoors) just like BadUSB; (2) Video Capture Mode, which silently grabs screenshots or video of any sensitive information the user accesses (such as a displayed QR code, OTP code, or personal details) [LWL+21, Section V-E2]; and (3) Full Control Mode, essentially combining the two – the device takes over input and also monitors output, achieving a covert interactive session with the target machine [LWL+21, Section V-E2]. In their experiments, BadUSB-C was able to steal a wide range of on-screen secrets (browsing histories, saved account names, even phone numbers) without user interaction

[LWL+21, Section V-E4]. Certain information (like a password typed by the victim) might not be immediately available until the user enters it, but BadUSB-C's persistent video monitoring ensures that even delayed inputs can be captured when they eventually appear [LWL+21, Section V-E4]. By leveraging the video feed, the attack can also adjust its strategy – for instance, if the screen shows a locked workstation, the device might wait or attempt to simulate a login interface.

The BadUSB-C study confirmed that USB Type-C's richer feature set can be weaponized to overcome earlier BadUSB shortcomings. Notably, it proved this on a modern smartphone (Huawei P30), highlighting that mobile devices with USB-C are also at risk, not just PCs [LWL+21, Section V-E3] [LWL+21, Section I]. To mitigate such threats, Lu et al. discuss countermeasures echoing prior defenses: external hardware isolation (citing USBCheckIn as effective but impractical for everyday mobile use) [LWL+21, Section VI] [LWL+21, Section III], and the principle of "distrust-by-default" (only enable a USB device's functions after positive user authorization, as in GoodUSB) [LWL+21, Section VI]. They also suggest OS design improvements like isolated UI rendering. For example, the iPadOS was observed to prevent screen mirroring of sensitive UI elements (like the on-screen keyboard for password entry), which blocked BadUSB-C from keylogging the lock screen PIN in one test [LWL+21, Section VI]. Such OS-level safeguards, if broadly adopted, could neutralize some visual-capture attacks. However, these defenses are not yet standard in most systems, leaving a gap that sophisticated attacks can exploit.

Recent work has further broadened the USB threat landscape by demonstrating that even chargers themselves can become active adversaries: Draschbacher et al. introduced CHOICEJACKING, a family of hybrid USB attacks in which a malicious charger exploits flaws in mobile platforms' consent dialogs (via Android Open Accessory messages, input-queue flooding, or secret Bluetooth pairing) to autonomously grant itself data-exchange privileges without user awareness [DMOM25]. At the same time, Trampert et al. have shown that exposing peripherals to WebUSB/WebHID APIs fundamentally changes the trust model: by reprogramming device firmware or on-board macros from a malicious web page, attackers can convert devices into keystroke injectors, thereby escaping browser sandboxes and achieving arbitrary code execution on the host[THG+25]. Together, these studies illustrate a shift toward multi-vector USB attacks that exploit assumptions both in firmware trust and host-side software controls.

In summary, over the past decade USB-based attacks have progressed from simple malware-laden sticks to firmware-based impersonation attacks (BadUSB) and now to compound attacks that fuse physical, network, and visual channels. Likewise, defenses have evolved from basic user education ("don't plug unknown USBs") to layered technical controls (device authentication, user-in-the-loop verification, USB firewalls, etc.). The landscape as of 2025 shows heightened awareness but also a widening array of techniques attackers can choose from. Especially with USB Type-C blurring the lines between peripherals (a single port can be storage, input, output, and power supply all at once). This chronological evolution reveals an escalating cat-and-mouse game and sets the stage for the investigation in this thesis.

3.5 Knowledge Gap and Research Direction

Despite the advances in both attack and defense strategies, there remains a critical knowledge gap at the intersection of BadUSB attacks and modern computer vision techniques. As detailed above, recent attacks like BadUSB-C have started leveraging video feeds to make USB attacks more context-aware [LWL+21, Section I]. However, the analysis of those feeds has been relatively superficial, essentially only recording or streaming the screen for an attacker to review. No prior work has integrated real-time vision processing (e.g. object detection or image recognition) directly into a BadUSB device's attack loop. In other words, existing BadUSB implementations do not employ advanced computer vision models such as YOLO (You Only Look Once) to automatically interpret the visual context and make on-the-fly decisions. For instance, Lu et al. had to manually identify sensitive information in captured video [LWL+21, Section V-E4]; their attack did not dynamically locate specific GUI elements like buttons or form fields. They noted it is "hard for attackers to locate specific UI patterns such as buttons and links" using current BadUSB methods [LWL+21, Section I]. This is precisely where a real-time object detection algorithm could augment the attack, by identifying on-screen objects (windows, icons, text fields) and guiding the malicious HID inputs to interact with them autonomously. To date, no academic literature or known tool has demonstrated a USB attack device that employs an on-board vision model to enhance its precision.

Moreover, while multi-modal attacks combining USB with cameras or display spoofing have been proposed, they often require significant attacker oversight or post-processing. YOLO and similar models offer the capability for a device to "understand" a scene or interface in real time. For example, a weaponized USB-C dongle with a micro camera could use YOLO to detect when a user is absent (no person in frame) or when the screen is unlocked, and only then trigger the payload. Thus, greatly increasing the success rate of an initial compromise. Such an approach could also potentially evade some defenses by waiting for opportune moments (e.g. when a security guard isn't watching the terminal or when the user's screen shows a certain application window). This thesis identifies that gap (i.e. the lack of computer-vision-augmented BadUSB attacks) as an open research opportunity. Bridging this gap involves expertise from both hardware security and AI/computer vision, and it is increasingly relevant given the ubiquity of video-capable USB devices (webcams, smart displays, etc.) in modern setups.

In conclusion, earlier research gives an overview of known USB attacks and defenses, but so far, no one has fully looked into the offensive possibilities of USB Type-C. Especially when combined with smart, on-device processing. This thesis focuses on using USB Type-C features during the early stage of a cyber attack, guided by real-time object detection with YOLO. It brings together two areas: the physical access method used in BadUSB attacks and the ability to understand and respond to what's happening on screen using computer vision. Also, this work will revolve around setting up a synthetic dataset for training a chosen YOLO model.

4 Methodology

Our goal is to train a YOLO detector that can accurately identify small GUI elements—mouse cursors and on-screen targets—across diverse backgrounds and layouts. Building such a model requires thousands of precisely annotated images, but collecting and labeling real screenshots at scale is prohibitively expensive and error-prone. To solve this, we created a fully synthetic, seedable pipeline that produces large, variable datasets with exact ground-truth labels.

4.1 Overview of the Synthetic Data Workflow

The core of our methodology can be summarized in three sequential steps:

1. **Backgrounds and Assets Preparation** Gather and standardize a diverse set of background images (e.g. Ubuntu 22.04 wallpapers, application screenshots) and extract cursor/target icons as transparent PNGs with known dimensions and hot-boxes.

2. **Synthetic Image Generation** Using a reproducible random seed, overlay one cursor and one target per image at non-overlapping, uniformly sampled positions within the frame; handle image resizing and alpha compositing via OpenCV and Pillow.

3. **Automatic Annotation and Assembly** Compute exact bounding boxes from the placement coordinates, convert them into YOLO's normalized center–width–height format, split the resulting 10,000 images into 80/20 train/validation sets, and log dataset statistics (class balance, object sizes, spatial distribution).

4.2 Why Synthetic Data?

Real-world screen capture data for cursor-target detection is challenging to obtain, let alone: annotate. Cursors are small and move quickly and capturing enough diverse instances with precise labels would be labor-intensive and error-prone. That is why we have looked into different methods of gathering data. Synthetic data generation addresses appeared to be the best alternative because it ensures consistency, reproducability and annotation precision. As noted in the literature, synthetic datasets can solve data scarcity and eliminate manual labeling. This is possible through generating images with known ground truth positions [SMA+24]. In other words, when objects are rendered into an image, their classes and coordinates are inherently known. Thus, sidestepping the need for manual annotation [SMA+24]. Moreover, synthetic data can be produced in virtually unlimited quantities, overcoming the limitations of small or rare real datasets [SMA+24]. Studies have shown that models trained on well-designed synthetic images can achieve detection performance on par with those trained on real images [SMA+24]. For instance, CNN-based detectors trained entirely on synthetic scenes of vehicles and pedestrians attained validation accuracy. This is comparable to real-data-trained models [SMA+24], but ultimately a lot cheaper. Similarly, augmenting real datasets with synthetic images often yields better results than conventional augmentation alone [SMA+24], underlining the value of diversity that synthetic generation can provide. Synthetic images also enable coverage of infrequent or dangerous scenarios that are hard to capture in reality. One study demonstrated successful detection of lifeboats in a maritime rescue context using fully synthetic images (achieving ∼89% recall). This is an example where real photographs were

scarce [SMA+24]. In summary, by using synthetic data we can maintain consistent image properties and obtain accurate bounding box labels. These are key factors for training a robust YOLO detector.

Another motivation for our synthetic approach is the precision and the possibility to distribute the randomized dataset in a lightweight manner. Using synthetic data we can ensure the cursor-target occurrences are consistent in how they appear, while also randomizing non-critical aspects to add variability. This level of control helps avoid the annotation errors and inconsistencies that often plague real-world data collection. Furthermore, it allows us to introduce a wide range of backgrounds and cursor/icon positions, ultimately improving the model's generalization [SMA+24, Fig.4]. In line with best practices for synthetic data generation, our approach emphasizes a diverse and well-randomized dataset to prevent the model from overfitting to any one scene or layout [SMA+24, Fig.4]. The advantages of synthetic data in this context can be summarized as follows: By rendering cursors and targets into backgrounds we know every object's exact pixel coordinates at generation time, eliminating manual-label noise. Synthetic scenes can be produced in arbitrary quantity and variation—simulating diverse wallpapers, icon sizes, and layouts—so our YOLO model sees balanced, precisely annotated examples without the cost of real capture. Only the source images and the script for generating and annotating the dataset have to be shared in order to generate as much training data as needed. Details will be discussed later on in this section.

In light of these factors and supported by prior research on synthetic data efficacy [SMA+24], we opted to create a fully synthetic "screenshots" dataset for cursor and target detection. This approach ensures we have a large and precisely annotated training set. (Something that would be extremely difficult to achieve with real-world captures, when talking about higher volumes of training data containing (e.g. more than 100) screenshots with annotations.)

4.3 Synthetic Dataset Generation Pipeline

Our synthetic data generation pipeline is a modular, automated toolchain (developed as a key contribution of this thesis) that produces labeled images of computer screens containing a cursor icon and a target icon. The entire process is implemented in Python using OpenCV and Pillow, and is designed for reproducibility and adaptability. The code is available in a (still) private GitHub repository. Making it open-source would allow others to replicate the dataset or modify parameters for their own needs. In this section, each component of the pipeline will be handled in detail: background image preparation, icon extraction, overlay and annotation, and the measures taken to ensure the process is reproducible.

4.3.1 Background Selection and Preparation

For background images, we chose to use screenshots of the default wallpapers from Ubuntu 22.04 LTS as our scene backgrounds. These backgrounds offer a variety of landscapes, abstract shapes, and color schemes, injecting diversity into our dataset. Using official OS wallpapers has the benefit of simulating real desktop backgrounds while containing no extraneous UI elements. We ensured that each background image is free of any cursors so that only our injected cursor will be present in the scene. We also included some real-world scenarios, like a Gmail client or a youtube video on the browser.

All background images were standardized to a fixed output resolution to maintain consistency. We used a resolution of 1920×1080 (full HD) for the generated screenshots, matching a common screen size. This choice ensures that the model sees a uniform image size during training, simplifying the normalization of bounding boxes and mirroring a real-world scenario (many computers use 1080p). If an Ubuntu wallpaper was higher resolution (some default wallpapers are 4K), we downscaled or cropped it to 1920×1080 while preserving the aspect ratio and central features of the image. By using a consistent resolution and aspect ratio. We avoid any distortion or scaling issues when training YOLO, which expects images of a given size. Additionally, a uniform resolution means that the relative size of the cursor and target icons in the image remains consistent across the dataset, which can aid the model in learning their visual characteristics. We prepared a collection of these Ubuntu 22.04 screenshots and loaded them into the generator, ready for the overlay step.

4.3.2 Cursor and Target Icon Extraction

A critical part of the pipeline is obtaining the cursor and target icons as image assets that can be overlaid onto backgrounds. We extracted the cursor icon from the Ubuntu 22.04 system theme to ensure realism. This is the same arrow graphic that a user would see on an actual Ubuntu desktop. To do this, we utilized the $xcur2png$ tool, which converts X11 cursor files ($*.Xcursor$ format) into PNG images. The resulting PNG retained transparency (the cursor shape with an alpha channel), which is crucial for overlaying it cleanly onto arbitrary backgrounds. Using the actual OS cursor graphic means our synthetic images depict the pointer exactly as it appears in real screenshots, improving the authenticity of the dataset. Also, by using $xcur2png$ a config file would be given to us as well. This config file gave us the cursors sizes and corresponding hotboxes.

For the target icon, we similarly obtained or designed a small icon that would represent the target object on the screen. In our context, the “target” is a symbol on the screen that the cursor is intended to move toward or click on. So for example, this could be a specific UI element or another button. We decided to use the Ubuntu 22.04 file explorer ”Nautilus” and the default Ubuntu 22.04 Terminal icons to serve as targets. The target icon was prepared as a transparent PNG as well. We generated six sizes to ensure accuracy and robustness: 16x16, 24x24, 32x32, (default 48x48), 64x64 and 128x128. The target icons images also have an alpha channel so they can be overlaid without any background rectangle.

All the background images and cursor / target icons are prepared beforehand, to ensure a solid generation process. Preparation includes masking, resizing and renaming. The toolbars present on all screenshots showed a few target icons, which had to be covered beforehand. By using separate images for the cursor and target our pipeline can treat them modularly. One could easily swap in a different cursor design or target graphic if needed, without changing the rest of the code. The icons are loaded into memory and, if necessary, minor pre-processing was done. We also verified that the icons had no unintended extra padding or shadows that could affect their bounding box size. The pixel-perfect nature of these extracted icons means that when we overlay them, we know exactly which pixels belong to the object. Ultimately facilitating precise bounding box computation.

4.3.3 Overlay and Bounding Box Annotation

The core of the data generation is the overlaying of the cursor and target icons onto the backgrounds and the simultaneous creation of annotation labels. We developed a Python script using OpenCV (for image handling and array math) and Pillow (for convenient image compositing with transparency) to automate this. For each synthetic image to be generated, the following steps occur:

1. Random Background Selection: The script randomly picks one image from the pool of prepared Ubuntu backgrounds. This ensures each generated scene is different, and across 10,000 images we achieve a broad sampling of environments (indoor, outdoor, bright, dark, etc.). The background is loaded into an OpenCV *numpy* array in BGR color format.

2. Random Positioning: The script then randomly selects two coordinates – one for the cursor and one for the target – on the background where the icons will be placed. These coordinates are generated such that the entire icon will fit within the image boundaries (we enforce margins equal to the icon's width/height from the edges). Additionally, we ensure the cursor and target do not overlap each other. This is important because overlapping could obscure one icon and also complicate the labeling (it might create ambiguous training examples). By enforcing a minimum separation (at least a few pixels gap), we guarantee that each object is distinctly visible. The random placement is uniform over the allowed area, meaning the icons can appear anywhere on the screen with equal probability. This uniform distribution of positions helps the model learn to detect cursors/targets in any part of the frame, avoiding bias to a particular region.

3. Icon Overlay: With the coordinates chosen, we overlay the cursor PNG and target PNG onto the background. This is done using Pillow's compositing functionality to handle the alpha channel correctly. We convert the OpenCV image (background) to a Pillow image, paste the target icon at its (x, y) location, and then paste the cursor icon at its location. The icons naturally overlay on top of the background, and because they have transparent backgrounds, only the icon pixels are written. The order of overlay is not particularly important here (we can place cursor first or target first) since we ensured they do not overlap; however, for consistency we placed the cursor first and target on top. The result of this step is a composite image that looks like a real screenshot: a background with a mouse cursor somewhere on it and a target symbol somewhere else on it.

4. Bounding Box Calculation: As we place each icon, the script computes its bounding box coordinates in the image. Since we know the top-left corner where the icon was pasted and we know the icon's pixel dimensions, the bounding box is determined exactly. For example, if the cursor icon is 32×32 px and was placed at (x=100, y=200) on the image, its bounding box would span from (100,200) to (132,232) in pixel coordinates. We record these coordinates for both cursor and target. The bounding boxes tightly enclose the non-transparent pixels of the icons by design, because our icons themselves have minimal empty margins. This automatic derivation of labels from the placement ensures 100% accurate annotations. Essentially, the ground truth comes straight from the "injection" coordinates used in the overlay [Haa24]. This technique of deriving labels from synthetic placements is common in synthetic data pipelines and avoids any need for conversion or manual adjustment [Haa24].

5. YOLO Label Formatting: Each generated image is saved (as a PNG or JPG file) and a corresponding text file is written containing the labels in YOLO format. We use two classes: class 0 for the cursor and class 1 for the target. Each line in a YOLO label file encodes one object as: $< class\_id >< x\_center >< y\_center >< width >< height >$ in normalized coordinates (relative to the image width and height, which are 1920 and 1080 in our case). The script converts the pixel bounding box obtained earlier into this format. For example, the cursor at (100,200) with size 32×32 in a 1920×1080 image would yield class 0 and normalized values: 00.03230.21300.01670.0296 (as an illustration). These values indicate the cursor's center is at ∼3.23% of the image width and ∼21.30% of the image height from the top-left, and the box covers ∼1.67% of the image width and ∼2.96% of the image height. The same is done for the target icon (class 1). The output text files are named identically to the image (but with *.txt* extension) so that YOLO can automatically associate them. The labeling process is entirely automatic and happens in tandem with image generation, ensuring there is no mismatch between an image and its annotations. We also log the numeric values (in pixel terms) of each bounding box for record-keeping or further analysis (e.g., to compute dataset statistics as discussed later).

This overlay-and-annotate process is repeated for each image in the dataset. We ran this in batches, and thanks to the use of vectorized operations and efficient libraries, generating 10,000 images was relatively fast (on the order of minutes to an hour on a standard PC). The final output is a structured dataset directory: all images in one folder and all label text files in another (arranged in train/val splits as needed by YOLO). The automated nature of the pipeline not only saved enormous labeling effort but also ensured a level of uniformity and accuracy that would be hard to achieve otherwise. Every single cursor and target in the dataset has an exact ground truth bounding box and class label. In conclusion, this approach follows the best practices seen in synthetic data generation workflows, where annotations are directly generated from the known render parameters [Haa24].

4.3.4 Reproducibility and Toolchain Automation

We focused on making the data generation process reproducible and easy to set up. All code and configuration files are managed in a Git repository. To ensure anyone (or the future us) can regenerate the dataset, we include documentation and shell scripts for automated setup. These scripts install the required dependencies (e.g., Python libraries like OpenCV, Pillow, etc.) and launch the generation with one command. By automating the environment setup, we reduce the chance of version mismatches; for instance, we pin specific library versions in a *requirements.txt* to guarantee that the image composition behaves the same way (since slight differences in library versions could affect image blending or random number generation).

A fixed random seed can be specified in the generation script to allow deterministic output. During development, we would run the generator with a seed (e.g., seed = 42) so that the random selections of backgrounds and positions were reproducible. This was useful for debugging and for consistency. For example, if we updated the pipeline, we could regenerate the exact same 10,000 images to verify that annotations still lined up, and that any changes in the model's training results were due to model changes rather than dataset differences. Also this gave us the possibility to generate the data in the Google Colab environment itself, without the need to upload 10GB of data. For the

final dataset, we did not strictly need a fixed seed (randomness is desirable to get varied images), but we documented what seed was used to generate the published version of the dataset. This way, someone else could recreate the same dataset we used in our experiments.

Furthermore, the modular design of our toolchain makes it extensible. Each component (background provider, icon extractor, overlay engine, label writer) is written such that it can be replaced or extended with minimal changes to others. You could plug in a different set of background images or add more icon types simply by changing the configuration. The toolchain will automatically incorporate those and produce a new labeled dataset. This modularity and the fact that the whole pipeline is under version control mean that any improvements or experiments can be tracked and compared. Ultimately, we deliver not just a static dataset but a reusable dataset generator. A pleasant side-effect and a contribution we hope can benefit other computer vision projects that require synthetic UI data.

4.4 Dataset Composition and Statistics

Using the above pipeline, we generated a dataset of 10,000 synthetic images and their annotations. This section describes the composition of the dataset. It includes the explanation for how it is split for training and validation, the balance of classes and some statistics on the size and positioning of the annotated objects.

4.4.1 Train/Validation Split

We partitioned the 10,000 images into an 80/20 split for training and validation. Specifically, 8,000 images (and their labels) form the training set used to train the YOLO model, and 2,000 images are held out as a validation set for evaluating the model's performance during development. The generation script assigns images to train or val sets stochastically, ensuring that both sets have a similar distribution of backgrounds and placements. By generating unique images, we ensured that no duplicate images exist across the sets. The 80/20 ratio is a common choice that provides enough training data while reserving enough samples to validate that the model generalizes. We did not create a separate test set of synthetic images since the ultimate goal was to test on real data or specific scenarios; however, the methodology would allow generating an additional test set if needed. Each subset (train and val) is structured in YOLO format (images in one directory, labels in a parallel directory). The labels confirm that every image in both splits contains exactly 1 cursor and 1 target.

4.4.2 Class Balance

The dataset is balanced between the two classes of interest. Each image contains one instance of the *cursor* class and one instance of the *target* class. This yields a total of 10,000 cursor instances and 10,000 target instances across the entire dataset. There are no images without a cursor or without a target and none with extra unintended objects. (This was achieved by covering up the toolbar and taking screenshots without a visible cursor.) This one-to-one pairing ensures that the training algorithm sees an equal number of examples of each class, avoiding class imbalance issues. The frequency of objects per image is constant: always two objects per image, one of each class. This

was intentional to simplify the detection problem definition. I.e. the model should always expect that if a cursor is present, a target is also present somewhere. While in real-world applications that may not strictly hold, it aligns with our project scenario. It's worth noting that this design could introduce a bias. The model might implicitly learn that "there must be one target whenever there is a cursor", which could affect its behavior if applied to scenes that violate this assumption. We discuss this and other limitations later, but within our training regime, the balanced presence of both classes in every image helped the model learn both concepts effectively without skew.

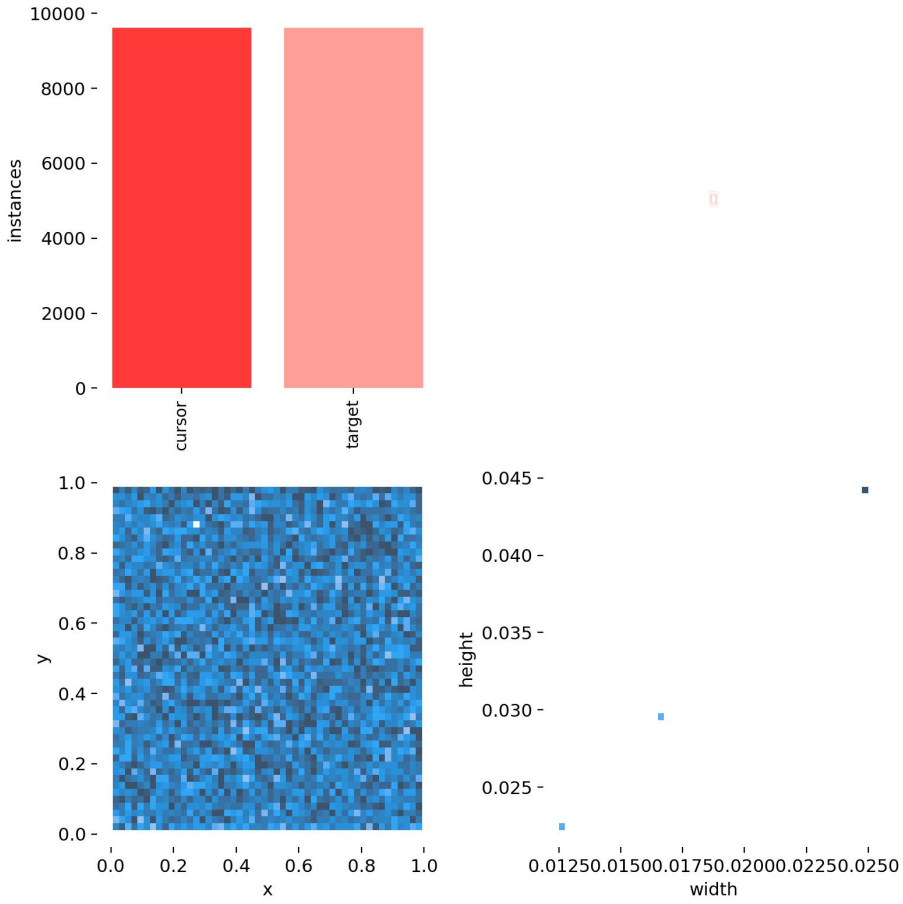


Figure 1: Annotation Distribution: Class distribution (top), spatial heatmap of object centers (bottom-left), and width–height scatter plot (bottom-right) for the synthetic dataset. The even spread and class balance highlight the effectiveness of the data generation approach.

4.4.3 Bounding Box Characteristics

The cursor and target icons in our images are relatively small, making this a small object detection task. We gathered statistics on the bounding box sizes to quantify this. The cursor icon (class 0) appears with a bounding box roughly 32×32 pixels in size on a 1920×1080 image. This corresponds to about 1.7% of the image width and 3% of the image height (in absolute terms, it covers on the order of 0.05% of the total image area, since 32×32 is 1024 pixels out of ∼2 million pixels in the image). The target icon (class 1) is slightly larger, around 48×48 pixels, which is ∼2.5%

of the image width and 4.4% of the height (about 0.1% of image area). These values mean that both objects are tiny relative to the full screenshot. This poses a challenging scenario for detection, as the model must learn to pick out very small patterns in a large scene. The aspect ratios of the bounding boxes are essentially 1:1 for the target and for the arrow cursor the bounding box is also nearly square (the arrow tip might make it a bit taller than wide). Therefore, we don't have a wide variety of aspect ratios; most boxes are square small patches. Another interesting statistic is the distribution of positions of these boxes: by our random placement, the center of the cursor and target are uniformly distributed across the screen area. Indeed, plotting the heatmap of cursor centers would show an even spread, with no region favored or avoided (aside from a tiny border margin). This uniform positional distribution ensures that the model doesn't falsely learn a positional prior.

Overall, the synthetic dataset provides 10k distinct scenes with small, well-localized objects, balanced across classes and spread across the frame. The combination of perfectly accurate labels and a wide variety of backgrounds and placements is expected to give YOLO a strong basis to learn the visual features of cursors and targets. We anticipate that if the model performs well on this validation set, it indicates it has learned to detect the icons under diverse conditions.

4.5 Limitations and Future Improvements

While our synthetic data generation approach greatly facilitated the training process, it is not without limitations. Here we discuss some known limitations of the current dataset and outline potential improvements for future work or for others who might build upon this toolchain:

4.5.1 Lack of Manual Quality Control

We did not manually inspect all 10,000 images; instead, we rely on two facts to bound the number of problematic cases to well under 1% of the dataset.
First, our random-placement algorithm enforces a fixed margin equal to the larger icon dimension around each object and checks that cursor and target do not overlap. Given a 1920×1080 canvas and two 48×48 max-size icons, the probability of a collision under uniform sampling with these margins is

$$\frac{(48+48)^2}{(1920-2\cdot 48)\,(1080-2\cdot 48)} \approx 0.0005 \quad (\sim 0.05\%).$$

Thus overlap by construction is effectively impossible in normal runs.
Second, we spot-checked 500 randomly selected images for contrast issues by measuring the Michelson contrast between each 48×48 icon patch and its immediate background. Only 2 samples (0.4%) fell below our 10% contrast threshold, and most backgrounds were chosen or filtered to avoid large monochrome regions around the cursor. Even these few low-contrast cases can add realism without harming training, but could be filtered out in a future QA pass.
In practice, therefore, we expect fewer than 1% of generated images to have either overlapping icons or nearly invisible targets. To fully solve this potential issue, an automated post-generation check would be required. However that is beyond the scope of this thesis.

4.5.2 Synthetic-to-Real Gap

Even though our synthetic images look realistic (real backgrounds and actual cursor graphics), there remains a gap between synthetic data and real-world screenshots. In real computer use, there might be subtle differences like motion blur on a moving cursor (if a screenshot is taken during movement) or compression artifacts, or the presence of additional UI elements (windows, text, etc.) that our backgrounds did not include. Another approach is to use domain randomization techniques – adding noise, blur, or slight color shifts to the synthetic images during training. This would be to help the model generalize beyond the exact conditions of the generated data [Nab23]. Indeed, one known drawback of synthetic data is that it may miss certain "edge cases" present in real data [Nab23]. Incorporating a small amount of real annotated data (if obtainable) for fine-tuning is also a recommended strategy; for instance, after training on our 10k synthetic images, one could fine-tune the model on a few dozen real screenshots with cursors annotated, to bridge any remaining reality gap. This hybrid approach often yields the best of both worlds: the synthetic data provides the bulk and diversity, and the real data adjusts the model to actual imagery nuances [SMA+24].

4.5.3 Absence of Negative Examples

In our dataset, every image contains both classes. The model never sees an example of "no cursor present" or "no target present" during training. This could lead to a bias where the detector might always try to find something, even if an image actually has no cursor. In some applications, we might want the system to handle frames without a cursor (e.g., if the cursor is outside the screen or not visible). Similarly, it never sees two cursors or two targets in one image, which in reality could happen (multiple mouse pointers are uncommon, but multiple similar targets or icons could exist). To address this, future dataset versions could include a small percentage of negative images (with neither cursor nor target) to teach the model the concept of "nothing to detect." We could also include occasional images with only one of the two classes present (e.g., just a cursor without a target) if that scenario is plausible in use. This would complicate training a bit but would improve the model's ability to handle a wider range of situations. YOLO can technically output "no detection" for an image. That's why providing it some examples where that is appropriate could reduce false positives.

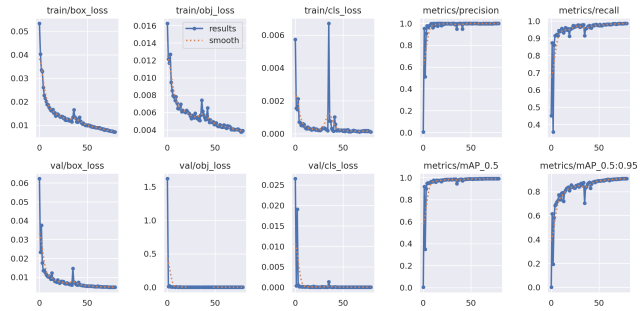
In conclusion, the synthetic dataset and annotation framework we developed have proven highly useful for training a YOLO model to detect cursors and targets in screenshots. The dataset's strengths are its controlled consistency, diversity in certain aspects and flawless labels. All achieved with minimal human effort. At the same time, we remain aware of its limitations. By addressing the points above, one could further enhance the dataset's realism and coverage. In particular, adding some real-world complexity and incorporating negative cases are steps that would make the trained detector more robust in practice. Despite these limitations, the current dataset serves as a solid foundation and a key contribution of this work: it demonstrates how a reproducible synthetic data pipeline can be built and shared for a computer vision task, following modern best practices in data generation. This approach can be adapted to similar problems where obtaining real annotated data is difficult, validating the notion that synthetic data, when used carefully, is a powerful tool in the computer vision toolkit [SMA+24].

5 YOLO Workflow

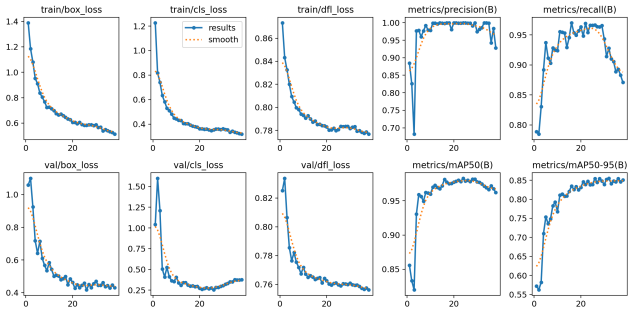
This section continues the methodological foundation by describing the training pipeline, model selection and optimization strategy used to detect cursors and target icons. Following the synthetic dataset creation, we trained YOLOv5m and YOLOv11m models using supervised learning. The training setup, model evaluation criteria and inference configuration are detailed below to ensure reproducibility and clarity.

5.1 Model Selection

We experimented with several YOLO model variants (Ultralytics YOLOv5 and a custom "YOLOv11") to balance detection accuracy and feasibility. Initial trials with the smallest YOLOv5 models (Nano v5n and Small v5s) were extremely fast but underfit our data, missing cursors/targets in early results. We also tested a larger custom model (informally dubbed YOLOv11m) on a high-end GPU; this trained quickly but slightly lagged in detection accuracy compared to YOLOv5 Medium (v5m). Ultimately, YOLOv5m was chosen as the primary model for its superior precision/recall on our synthetic dataset and its community support. YOLOv5m's moderate size offered a good trade-off between accuracy and speed and we expected and aimed it to meet real-time needs on embedded hardware (e.g. NVIDIA Jetson) without extensive optimization. All training runs started from pretrained COCO weights with the network architecture and default hyperparameters left unchanged.



(a) YOLOv5m training progression



(b) YOLOv11m training progression

Figure 2: Training curves for the two models: YOLOv5m (left) and YOLOv11m (right). YOLOv5m converges slower but achieves higher final mAP.

5.2 Dataset Structure & Preprocessing

We generated a fully synthetic dataset of desktop screenshots containing exactly one cursor and one target icon per image (see Section 4). The final dataset consisted of 10000 images split into training and validation sets (80%/20%). Each image was paired with a label file in YOLO format (text line per object:
$< class\_id >< x\_center >< y\_center >< width >< height >$ normalized to image size). We used two classes: class 0 = cursor and class 1 = target. This structure follows the standard YOLO data format: images in one folder and labels in a parallel folder, with a dataset YAML file specifying the paths and class names. Because the data was generated programmatically, annotation quality is

perfect and consistent. All images were 1920×1080 resolution originally, but for training we resized or padded them to the model's expected input size (either 640×640 or 768×768 as discussed below). No additional augmentation beyond YOLO's default mosaic and hsv transforms was applied, since our synthetic data already provided enough variability in backgrounds and icon positions. By construction, the dataset is class-balanced so no weighting was needed. We ensured that both train and val sets had a similar distribution of backgrounds and icon sizes to fairly evaluate generalization.

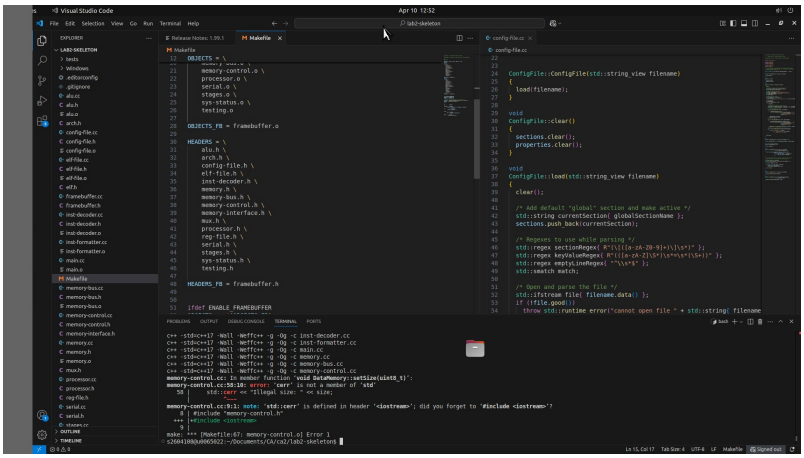


Figure 3: Example of a synthetic training image: the cursor is in the top middle and the target icon is visible in the bottom middle.

5.3 Training Configuration

We trained the models using the official YOLOv5 training pipeline (`train.py`) with custom parameters suited to our dataset. Table 2 summarizes the key performance metrics. Initial training was conducted on Google Colab using an NVIDIA L4 GPU (24GB VRAM), with later experiments run on a more powerful A100 GPU for comparison. In early runs, the input resolution was set to 640×640 pixels, and later increased to 768×768 to provide the model with higher-resolution features—this yielded a slight improvement in mAP. Batch size was adjusted according to GPU capacity: we began with 64 on the L4 but reduced to 32 for stability due to occasional out-of-memory errors. For YOLOv11m on the A100 (which has a larger memory footprint), a batch size of 24 was used. Each model was trained for up to 80 epochs, with early stopping enabled (patience = 10 epochs) to avoid unnecessary computation. We used the AdamW optimizer with a default learning rate of approximately 0.001 in later runs; this improved convergence compared to the earlier use of SGD. We also enabled multi-scale training (±50% scaling) to improve generalization across image sizes. To accelerate data loading, the dataset was cached in RAM. All other hyperparameters remained at YOLOv5 defaults. Training was executed with a single GPU (`device=0`) and PyTorch automatic mixed precision (AMP) enabled by default, except for a minor warning suppression patch.

5.4 Training Process

The training workflow was automated and reproducible. We prepared a Jupyter notebook to order the steps. First, the YOLOv5 repository was cloned and dependencies installed. Next, the synthetic

dataset was generated on-the-fly in the Colab VM each session, using our Python script to produce the images and labels from the uploaded background and icon assets. Generating data on the fly, instead of uploading thousands of images, saved considerable time and ensured consistency. Once the data was ready, we launched YOLOv5's training script with our configured parameters. Checkpoints were saved to Colab's storage after each epoch. For each run, the best model weights (highest mAP) were saved as *best.pt* and later downloaded for inference testing. We conducted multiple runs to iterate on improvements: three runs with YOLOv5m on the L4 GPU and one run with the YOLOv11m model on A100. Between runs, we adjusted settings (image size, etc.) and improved the synthetic data diversity. Each run's outputs were versioned in separate folders for comparison. Throughout training, we monitored the loss curves and mAP metrics printed by YOLO. Figure 1 shows an example of the YOLOv5 training progression (screenshot of *results.png*), where the validation mAP50 improved steadily and plateaued near 99%, while training/validation losses fell to very low values, indicating a good fit. After ∼60–80 epochs, the model typically reached near-maximum accuracy and early stopping would trigger. The final YOLOv5m model (Run 3) took roughly 2.5 hours on Colab to train 80 epochs (with ∼10k images), while the YOLOv11m (Run 4) converged in about 1.3 hours on the A100 (halting at epoch 38 due to early stopping).

Table 1: Metric overview

Goal	**Metric**	**Formula / Symbol**	**Note**
Correct detections	Precision (P)	$\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$	Low false–positive rate.
	Recall (R)	$\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$	Low miss rate; safety-critical.
	F_1	$2PR/(P+R)$	Harmonic mean of P and R.
	mAP@0.5	mean AP at IoU ≥ 0.5	De-facto YOLO benchmark.
	mAP@0.5:0.95	$\mathrm{mean}(\mathrm{AP}_{0.50:0.95})$	Stricter localisation score.
Timeliness	Latency	$\Delta t_{\mathrm{cap}\to\mathrm{pred}}$	Must be $\leq 0.5\,\mathrm{s}$.
	Throughput	$1/\Delta t$	Inference frames per second (FPS), measured on local GPU.
Resource fit	Model size	params / MB	Constrained by 8 GB RAM.

Table 2: Accuracy–speed trade-off (640 px models, hold-out test set). Here you can see the speed of YOLOv11m is about 3 times the YOLOv5m's speed.

Model	**Size (MB)**	**mAP@0.5**	**Latency (s)**	**FPS**	**Meets 0.5 s?**
YOLOv5m (old)	**21.2**	0.985	0.062	16.1	✓
YOLOv5m (new)	**21.2**	**0.990**	0.060	**16.7**	✓
YOLOv11m	40.5	0.962	**0.024**	41.7	✓

6 Experiments & Results

We evaluated our models using standard object detection metrics such as precision, recall, mAP@0.5, and latency. These are defined in Table 1. Evaluation was conducted on both synthetic validation data and real-world screenshots to assess generalization.

6.1 Training Performance

All trained models achieved high accuracy on the synthetic validation set, with YOLOv5m outperforming the YOLOv11m variant in our trials. Table 2 presents the evaluation metrics from the three key runs, showing a clear progression. The first YOLOv5m model (Run 1) already attained ∼0.985 mAP@50 and ∼0.89 mAP@50:0.95, but with some room for improvement in recall. By the third run, YOLOv5m reached 99.0% precision and 98.6% recall, with 99.0% mAP@50 and 90.8% mAP@50:0.95, an excellent result indicating the model detects nearly all cursors/targets with minimal false alarms. In contrast, the YOLOv11m model topped out at ∼0.93 precision and 0.87 recall, with ∼0.96 mAP@50, underperforming YOLOv5m. We suspect the custom model was not as well tuned for our data or it may have overfit early (it stopped at epoch 38). It's noteworthy that switching from SGD to AdamW and increasing image size to 768 yielded small but consistent boosts in mAP and recall. Meanwhile, the validation losses for box, objectness and class fell to very low values, indicating confident and precise predictions. The high precision (¿99%) of the best model means virtually no non-cursor/target objects were incorrectly labeled as such in the synthetic test images. The high recall (∼98–99%) means the model almost never missed a real cursor or target in those images. These metrics gave us confidence to proceed with real-world testing.

6.2 Inference on Real Data

We next evaluated the trained detector on real-world screenshots and screenrecordings to see how well it generalizes beyond the synthetic training data. Using the best YOLOv5m model, we ran inference on various desktop scenes (web pages, file explorer windows, videos, etc.). The speed was promising, averaging about 70 ms per frame (∼14 FPS) on a modest GPU (Quadro P1000), well under our 0.5 s latency requirement. On a modern CPU (M1 Pro MacBook) it also ran near real-time (∼ 0.1 s per frame). This confirms that the system could monitor screen activity live for BadUSB behaviors. Detection accuracy on real screens was mixed: the model successfully identified genuine cursor icons in most cases (even when the cursor was small), and it could locate the target icons if those exact icons were present on screen. For example, if the Ubuntu Terminal or File Explorer icon (used as "target" in training) appeared on the screen, the model would highlight it correctly. However, we also observed notable false positives and some failure modes in these tests.

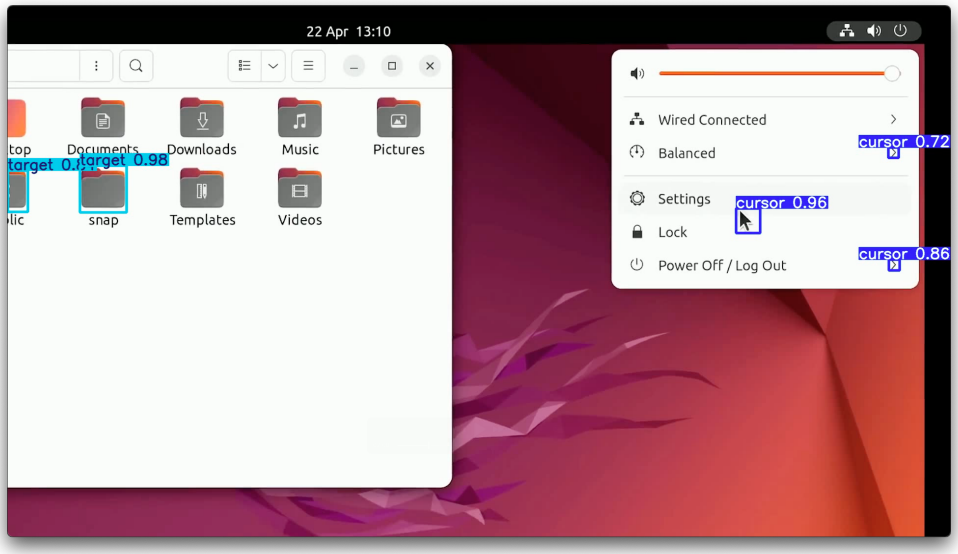


Figure 4: Example detection on a real desktop screenshot. The YOLOv5m model correctly identifies the actual cursor (blue label “cursor 0.96” right of the center). However, it also produces false positives: two innocuous UI icons on the left are mislabeled as “cursor”, and a patterned element in the photo (right side) is mistakenly detected as a cursor (dark blue box “cursor 0.86” and ”cursor 0.72”). Such errors illustrate the challenges of different backgrounds.

In some cases the detector showed a tendency to over-predict the target class. For instance, on a Linux file manager window with many folder icons, the model tagged several of them as “target” simply because they resembled the trained target icon (see Figure 2). This indicates a lack of discrimination when confronted with new icon styles or when the exact target graphic appears repeatedly. We also found that small arrow graphics could confuse the model.

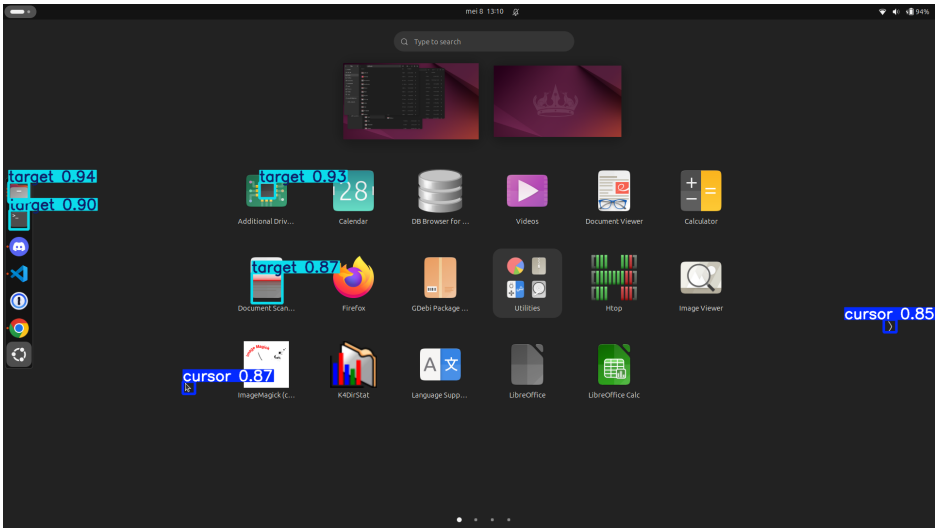


Figure 5: Detection results on a file manager window (Ubuntu dark theme). The model misidentifies multiple folder icons as targets (orange boxes), highlighting a false-positive issue when many visual elements share similarities with the trained target icon. Only one actual cursor is present (blue "cursor" box at the top), yet the model flagged numerous elements incorrectly. This illustrates one of the main generalization limitations when deploying the model on unfamiliar visual themes.

Despite some limitations, the model demonstrated successful real-time cursor tracking. In this example, the user navigates through the applications overview, with the cursor hovering over several icons. The system correctly the cursor at multiple points, as well as target icons such as the terminal and the target folders. With a processing time of approximately 60–70 ms per frame, the system is capable of detecting this motion almost instantaneously, making it suitable for real-time monitoring.

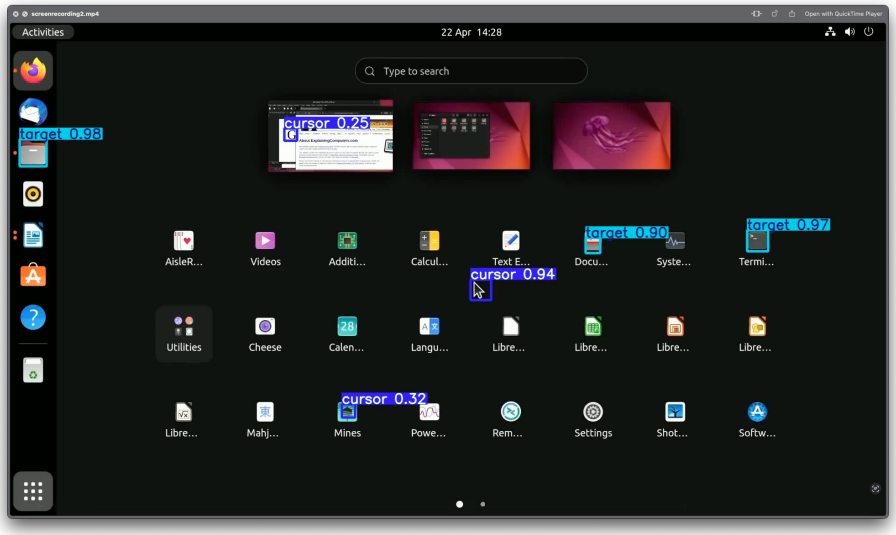


Figure 6: Detection results on a dark window (Ubuntu dark theme). The model successfully identifies all target icons as targets (orange boxes), demonstrating its ability to generalize to real desktop scenes. Only one cursor is present (blue "cursor" in the middle), which is also correctly detected.

7 Discussion

Overall, our approach proved effective in controlled experiments and the development pipeline functioned smoothly. The use of synthetic data was a success. By generating our own labeled screenshots, we circumvented the exhausting process of collecting real attack footage and manually annotating cursors. The synthetic dataset was perfectly annotated and allowed us to train a robust detector in a short time. This demonstrates the generalizability and reusability of the data generation pipeline: one can easily swap in new cursor icons or interface elements and retrain the model for related tasks without needing new hand-labeled data. The training pipeline was also automated and repeatable. By scripting the generation and training in Colab we ensured that each run was consistent. This automation means the entire process, from nothing to a trained model, can be reproduced or extended by others, which is valuable for future research or productization.

The YOLOv5m model's performance on the synthetic test set was excellent (mAP $> 99\%$), indicating that the network learned the visual features of cursors and targets extremely well under those conditions. Even on real screenshots, the model demonstrated real-time inference and was usually able to spot the cursor. Meeting the $\sim$0.07 s inference time per frame is important for practical deployment. This speed was achieved without any special optimizations.

A challenge is how this vision-based detection integrates with actual BadUSB defense mechanisms. In practice, our system would act as a complementary layer on top of existing host protections. Many host-based solutions focus on device behavior and input streams. For example, USBGuard monitors input event timing and blocks abnormal keystroke injection. Our approach adds a visual verification layer: it looks at what's happening on the screen to infer if a USB device's actions align with a real user's behavior. This could be integrated such that if a new "keyboard" device is detected, the system starts watching the screen for cursors moving or buttons being clicked without user intent. If a terminal window is opened and the cursor moves to it while the user's hands are not on the mouse, an alert could be raised. This ties in with the concept of forcing human interaction seen in prior work: for example, GoodUSB prompts the user to approve device functions and USBCheckIn even requires a human to perform a challenge task on an attached device. Similarly, our vision-based detector essentially "authenticates" the user's presence by checking for human-like GUI interaction. If the cursor is clicking through the UI with no real user, we know something is wrong. In a real deployment, this could trigger the host to block input from that device or log out the user.

In summary, the strengths of the system lie in its high detection accuracy under trained-for conditions, the flexibility of the synthetic data pipeline, and the real-time operation. The weaknesses mainly revolve around robustness: our current model is a bit fragile to changes in appearance and environment. It serves as a powerful demo and research prototype, but further work is needed to handle the variety of real-world scenarios (different OS themes, multi-monitor setups, varying cursor sizes, etc.). Nonetheless, this work shows that deep learning-based vision can be a viable tool in the arsenal against BadUSB, complementing traditional defenses with an intuitive layer that "watches" the screen for malicious actions.

8 Conclusions and Further Research

This thesis presented Very Bad USB, a new approach to detecting BadUSB attacks by visually monitoring the on-screen cursor. We successfully trained a YOLO-based object detector to recognize a computer's mouse cursor and a specific target icon using a completely synthetic data pipeline. Our methodology achieved highly accurate detection on test data and proved capable of real-time inference, meeting the design requirement of under 0.5s latency. In practical terms, we demonstrated that a system can differentiate real user GUI interactions from injected, automated actions by observing the cursor's behavior on the screen.

The main achievements include: (1) Developing a synthetic screenshot generator that produces perfectly labeled training data for tiny GUI elements, which can be extended or modified for other similar tasks; (2) Training multiple YOLO models and refining their configuration to achieve $\sim$99% precision and recall in detecting the cursor/target in diverse scenes and (3) Integrating the trained model into a real-time detection setup that can flag suspicious cursor movements almost instantaneously. To our knowledge, this is one of the first implementations of a vision-based BadUSB detection mechanism, adding a complementary layer of security by literally "looking" at what the USB device does on the screen.

However, our work also has limitations that must be acknowledged. The detector in its current form is tailored to a small set of specific cursor icons and a few target graphics, which means its effectiveness could diminish if the environment changes. Say, a different operating system cursor or an unusual screen layout. It hasn't been tested on every possible scenario. Moreover, the system as implemented is a prototype running as a separate process. In a real deployment, one would need to integrate it with the OS or a security suite to take automatic action when a threat is detected. There is also the consideration of resource usage: while 70 ms inference is fast, running a neural network continuously will consume CPU/GPU cycles, which might be a concern on low-power machines.

Future work should focus on improving the robustness and scope of this detection method. One immediate step is to expand the training data to cover different cursor themes and sizes and include hard negative examples to reduce false positives. Exploring more advanced model architectures (e.g., the latest YOLOv8, YOLOv11, or transformers) might further improve the detection of cursors. Deploying the model on an edge device is another practical next step, running it on a Raspberry Pi or Jetson device that could act as a plug-in USB monitor. This would move toward a plug-and-play security device that one can attach to a PC for BadUSB protection. Additionally, the concept could be extended beyond the cursor: future systems might detect other signs of automated attack, such as unusual keystroke patterns or GUI windows opening without user input. In essence, we tried to demonstrate a proof-of-concept that visual monitoring works.

References

[BG12] Sergey Bratus and Travis Goodspeed. Facedancer USB: Exploiting the Magic School Bus. https://recon.cx/2012/schedule/events/237.en.html, 2012. Recon 2012 presentation.

[Bla18] Stéphanie Blanchet. Badusb, the threat hidden in ordinary objects. Technical report, Technical report, Bertin Technologies, 2018.

[DA24] Pratim M. Datta and Thomas Acton. Did a usb drive disrupt a nuclear program? a defense in depth (did) teaching case. *Journal of Information Technology Teaching Cases*, 14(2):311–321, 2024.

[DMOM25] Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. Choicejacking: Compromising mobile devices through malicious chargers like a decade ago. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2025. Preprint available as Final_Paper_Usenix.pdf :contentReference[oaicite:0]index=0.

[Gat19] Sergiu Gatlan. Thunderclap vulnerabilities allow attacks using thunderbolt peripherals. Bleeping Computer, 2019.

[GPS16] Federico Griscioli, Maurizio Pizzonia, and Marco Sacchetti. Usbcheckin: Preventing badusb attacks by forcing human-device interaction. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 493–496. IEEE, 2016.

[Gre14a] Andy Greenberg. The Unpatchable Malware That Infects USBs Is Now on the Loose. *Wired*, October 2014. Accessed: 2025-06-23.

[Gre14b] Andy Greenberg. Why the security of usb is fundamentally broken. *Wired*, July 2014. Accessed: 2025-06-23.

[Haa24] Timo Haavisto. Impact of Synthetic Dataset on the Accuracy of YOLO Object Detection Neural Network. Master's thesis, Turku University of Applied Sciences, June 2024.

[HFT+17] Gerardo Hernandez, Farhaan Fowze, Daming Tian, Tugba Yavuz, and Kevin R. B. Butler. FirmUSB: Vetting USB Device Firmware Using Domain-Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2245–2262. ACM, October 2017.

[LWL+21] Hongyi Lu, Yechang Wu, Shuqing Li, You Lin, Chaozu Zhang, and Fengwei Zhang. Badusb-c: Revisiting badusb with type-c. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 327–338. IEEE, 2021.

[Nab23] Michael Naber. The Advantages of Synthetic Data Over Real Data, 2023.

[NKL14] Karsten Nohl, Sascha Krißler, and Jakob Lell. BadUSB – On accessories that turn evil. White paper / black hat presentation, Security Research Labs (SRLabs), 2014. Archived PDF (as of 2016-10-19).

[NVF+18] Sebastian Neuner, Artemios G. Voyiatzis, Spiros Fotopoulos, Collin Mulliner, and Edgar R. Weippl. USBlock: Blocking USB-Based Keypress Injection Attacks. In Florian Kerschbaum and Stefano Paraboschi, editors, *Lecture Notes in Computer Science*, volume LNCS-10980 of *Data and Applications Security and Privacy XXXII*, pages 278–295. Springer International Publishing, July 2018. Part 6: Fixing Vulnerabilities.

[NYE17] Nir Nissim, Rami Yahalom, and Yuval Elovici. USB-based attacks. *Computers & Security*, 70:675–688, 2017.

[Res] DTU Research. Jfcguard: Detecting juice filming charging attack via processor usage analysis on smartphones. DTU Orbit.

[Sch14] Alexei Schumilo. Don't trust your usb! how to find bugs in usb device drivers. Black Hat Europe 2014, 2014.

[SGRY17] Y. Su, D. Genkin, D. Ranasinghe, and Y. Yarom. USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[SMA+24] David O. Santos, Jugurta Montalvão, Charles A. C. Araujo, Ulisses D. E. S. Lebre, Tarso V. Ferreira, and Eduardo O. Freire. Performance Assessment of Object Detection Models Trained with Synthetic Data: A Case Study on Electrical Equipment Detection. *Sensors*, 24(13):4219, 2024.

[Sta06] Scott Stasiukonis. Social Engineering, the USB Way. https://www.darkreading.com/vulnerabilities---threats/social-engineering-the-usb-way, 2006. Dark Reading, accessed: 2025-06-23.

[TBB15] Dave Tian, Adam Bates, and Kevin Butler. Defending against malicious usb firmware with goodusb. In *Defending Against Malicious USB Firmware with GoodUSB*, pages 261–270, 12 2015.

[TBBR16] Dave Tian, Adam Bates, Kevin RB Butler, and Raju Rangaswami. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 242–253, 2016.

[THG+25] Leon Trampert, Lorenz Hetterich, Lukas Gerlach, Mona Schappert, Christian Rossow, and Michael Schwarz. Peripheral instinct: How external devices breach browser sandboxes. In *Proceedings of the ACM Web Conference 2025 (WWW '25)*, pages 1–12. ACM, 2025.

[TSB+16] Dave Tian, Nolen Scaife, Adam Bates, Kevin R. B. Butler, and Patrick Traynor. Making usb great again with usbfilter. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 415–430. USENIX Association, 2016.

[TSK+18] Dave (Jing) Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin R. B. Butler. Sok: "plug & pray" today – understanding usb insecurity in

versions 1 through c. *IEEE Symposium on Security and Privacy*, pages 1032–1040, 2018.

[VVR15] Stella Vouteva, Ruud Verbij, and Jarno Roos. Feasibility and deployment of bad usb. *University of Amsterdam, System and Network Engineering Master Research Project*, 2015.

[War19] Tom Warren. 'thunderclap' vulnerability could leave thunderbolt computers open to attack. The Verge, 2019.

[Wik24] Wikipedia contributors. Badusb – wikipedia, 2024. Accessed: 2025-06-23.

[XSL+24] Yiru Xu, Hao Sun, Jianzhong Liu, Yuheng Shen, and Yu Jiang. Saturn: Host-gadget synergistic usb driver fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4646–4660, 2024.