Development of a Raspberry Pi-Based Secure USB Passthrough System to Mitigate Bad USB Keystroke-Injection Attacks

Muhammad Alif Nukman bin Nor Azman[1], *Nurhashikin Mohd Salleh[2], Siti Rahayu Selamat[3], Aimi Liyana Amir[4], Mohd Taufik Mishan[5]

[1,2,3]Faculty of Artificial Intelligence and Cyber Security, Universiti Teknikal Malaysia Melaka, Hang Tuah Jaya, 76100 Durian Tunggal, Melaka, Malaysia.

[4,5]Faculty of Computer and Mathematical Sciences, Universiti Teknologi Mara, Cawangan Melaka, Kampus Jasin, Jalan Lembah Kesang 1/1-2, Kampung Seri Mendapat, 77300 Merlimau, Melaka, Malaysia.

***Corresponding Author**

DOI: https://doi.org/10.47772/IJRISS.2026.100800329

Received: 19 August 2026; Accepted: 24 August 2026; Published: 03 September 2026

ABSTRACT

BadUSB attacks exploit the implicit trust that operating systems place in USB Human Interface Devices (HIDs), enabling a malicious peripheral to inject automated keystrokes and execute commands as if they originated from a legitimate user. This study develops a low-cost USB passthrough system that operates independently of host-side security software, using a Raspberry Pi 4 as an intermediary security layer between a keyboard and a host computer. The prototype combines VID/PID-based device identification, whitelist and blacklist controls, real-time keystroke-timing analysis, CAPTCHA-based human verification, input forwarding, and security-event logging. A Raspberry Pi Pico configured as a malicious HID was used in a preliminary, attack-driven functional evaluation comprising seven black-box test cases covering device enrolment, normal keyboard passthrough, resilience, malicious-keystroke detection, human verification, logging, and administrative functions. Of 18 predefined functional outcomes, 16 passed, one partially passed, and one failed. These outcomes demonstrate the functional feasibility of the prototype under the tested configuration but do not represent detection accuracy or general classification performance. The limited evaluation did not support the calculation of a confusion matrix, precision, recall, F1-score, or false-positive rate. The preliminary latency comparison indicated an additional average delay of 1.15 ms; however, the available experiment did not provide sufficient statistical evidence for broader performance interpretation. Broader repeated testing involving multiple users, devices, operating systems, attack patterns, and statistically rigorous latency measurements is required before operational deployment.

Keywords: BadUSB, Human Interface Device (HID), Raspberry Pi, USB Passthrough, Keystroke Injection, Behavioural Detection

INTRODUCTION

Universal Serial Bus (USB) technology is widely used because it provides a convenient and standardized interface for peripherals, storage devices, and communication hardware. That convenience also creates a security problem: host operating systems generally trust a connected device according to the function declared by its USB descriptors. A device that identifies itself as a keyboard can therefore generate input immediately, frequently without explicit user authorization. BadUSB attacks exploit this trust by modifying device firmware or configuring a microcontroller to emulate a legitimate peripheral while performing malicious actions [1], [2], [3], [4]. Malicious HID attacks are particularly difficult to control because keyboards are essential input devices and cannot be treated in the same way as removable storage. A device can inject a sequence of keystrokes that opens a command interpreter, modifies system settings, downloads a payload, or creates persistence. To the operating system, these actions can appear similar to input from an authorized user. Host-based antivirus and

storage-control policies may therefore provide limited protection when the attack is delivered through a standards-compliant HID interface [5], [6].

Existing countermeasures include static device authorization, host-side USB policy enforcement, firmware-aware mediation, USB proxy architectures, behavioral monitoring, and machine-learning methods [7], [8], [10], [11], [12]. Each addresses part of the problem. Static VID/PID filtering may be defeated by identifier spoofing; host-side tools depend on operating-system configuration; and behavioral approaches may incorrectly classify rapid legitimate typing as automated input. These limitations motivate a layered design that prevents the peripheral from connecting directly to the protected host and combines VID/PID-based device identification, whitelist-based access control, behavioural inspection, user verification, and logging. This study develops a secure USB passthrough prototype using a Raspberry Pi 4. The Raspberry Pi operates as a USB host toward the connected keyboard and as a USB HID gadget toward the protected computer. Input events are intercepted, evaluated, and forwarded only after the device and its behavior satisfy the implemented controls. The contribution is not a new USB protocol or a universal BadUSB detector. Instead, it is a practical integration of several defensive controls into a low-cost, portable prototype that operates independently of host-side security software, together with a functional evaluation using a physical malicious-HID test device.

The remainder of this paper is organized as follows. Section 2 reviews BadUSB attacks and related countermeasures. Section 3 describes the proposed architecture, detection logic, implementation, and evaluation method. Section 4 presents the functional and latency results. Section 5 discusses the findings, limitations, and future research directions.

RELATED WORK

This section reviews the characteristics of BadUSB and HID-based keystroke-injection attacks, followed by existing defensive approaches for detecting or mitigating malicious USB devices. It also identifies the research gap addressed by the proposed secure USB passthrough system.

BadUSB and HID-Based Keystroke Injection

The USB protocol supports multiple device classes, including mass storage, network adapters, audio devices, and HIDs. During enumeration, the peripheral reports descriptors that inform the host of its claimed functions. Conventional systems commonly accept these descriptors without cryptographic proof that the connected hardware is behaving according to the user's expectation [1], [13]. This trust boundary permits a reprogrammed or purpose-built microcontroller to masquerade as a keyboard and submit commands at machine speed.

Nissim et al. [4] developed a taxonomy of USB-based attacks and emphasized that a peripheral can move between data, electrical, and device-emulation attack surfaces. Pham et al. [13] similarly reviewed USB software attacks and noted the difficulty of enforcing security when devices can expose several interfaces. Practical work using Arduino-compatible boards and Rubber Ducky-style devices has shown that inexpensive hardware can execute HID injection attacks across common operating systems [3], [5], [14]. Recent studies continue to demonstrate that malicious HIDs can bypass multiple conventional layers because their input is processed as trusted keyboard activity [6].

Existing Defensive Approaches

Defensive mechanisms may be grouped into device authorization, host-based control, hardware or firmware mediation, and behavioral detection. GoodUSB introduced user-informed authorization of device functions to reduce the risk created by unexpected USB interfaces [7]. USBIPS proposed a host-protection framework for malicious peripherals [8], while USB proxy research demonstrated that an intermediary could inspect or redirect USB communication before it reaches the target [9]. These studies support the principle of placing a mediation layer at the USB trust boundary. Behavioral techniques analyze characteristics such as keystroke rate, interval regularity, command patterns, or power consumption. A neural-network-driven power-analysis method has been used to distinguish malicious USB peripherals without relying only on descriptors [10]. Behavioral methods can improve visibility when identifiers are spoofed, but their reliability depends on representative data and calibrated

thresholds. A sophisticated attack can intentionally introduce human-like delays, while a legitimate user may type or press keys rapidly enough to trigger a rule.

Table 2.1: Comparison of selected BadUSB countermeasures

Approach	Primary control	Strength	Limitation
GoodUSB [7]	User authorization of device functions	Addresses unexpected USB functionality	Requires user involvement and host integration
USBIPS [8]	Host protection framework	Centralized inspection and policy enforcement	Host-side deployment dependency
USB Proxy [9]	Intermediary USB mediation	Separates the peripheral from the protected host	Implementation complexity and device compatibility
Power-analysis method [10]	Neural-network behavioural classification	Can detect anomalies beyond VID/PID	Requires training data and specialized measurements
Proposed prototype	VID/PID identification, whitelist-based access control, timing analysis, CAPTCHA, and logging	Low-cost, portable, and independent of host-side security software	Threshold false positives and limited device coverage

The research gap addressed in this work is therefore the integration of VID/PID-based device identification, whitelist-based access control, keystroke-behaviour analysis, interactive human verification, and event logging in a single physical passthrough device. Unlike purely host-based control, the prototype prevents the unverified peripheral from communicating directly with the protected computer. Unlike a purely static whitelist, it continues to inspect the timing of input from an authorized device.

METHODOLOGY

This section describes the system architecture, layered security controls, prototype implementation, and evaluation procedures used to develop and test the secure USB passthrough system.

System Architecture

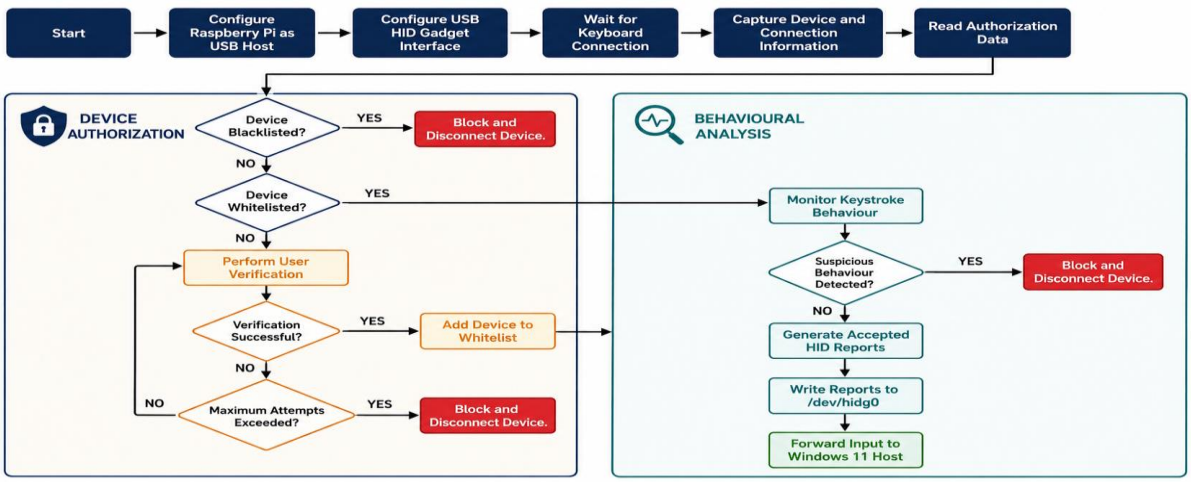

Figure 3.1: Device-identification, access-control, and behavioural-analysis workflow of the secure USB passthrough system.

The prototype was developed through an iterative system-development process comprising requirements analysis, architectural design, implementation, functional testing, and refinement. The system was built around a Raspberry Pi 4 Model B running Raspberry Pi OS. Its USB-A interface receives input from either a physical keyboard or a malicious HID device, while its USB-C On-The-Go interface operates in gadget mode and presents the Raspberry Pi as a keyboard to the protected Windows 11 host. A Python application captures input events through the Linux evdev interface, evaluates the connected device's identification and access-control status together with its input behaviour, and writes accepted HID reports to /dev/hidg0.

Figure 3.1 presents the device-identification, access-control, and behavioural-analysis workflow of the proposed system. During initialization, the Raspberry Pi is configured as a USB host for the connected keyboard and as a USB HID gadget for communication with the protected Windows 11 host. When a device is connected, the system captures its VID/PID and connection information and compares these details with the stored whitelist and blacklist records. A blacklisted device is immediately blocked and disconnected. If the device is not blacklisted, the system determines whether it has been whitelisted. A whitelisted device proceeds directly to keystroke monitoring, whereas an unknown device must complete CAPTCHA verification before being added to the whitelist. Repeated verification failures result in the device being blocked and disconnected.

During behavioral analysis, the system examines the timing characteristics of incoming keystrokes. Suspicious timing patterns are recorded, and CAPTCHA verification is triggered when the accumulated suspicious-activity count reaches the configured threshold. Successful verification allows monitoring to continue, whereas repeated CAPTCHA failures cause the device to be blocked and disconnected. Input that passes the security checks is converted into valid HID reports, written to /dev/hidg0, and forwarded to the protected Windows 11 host.

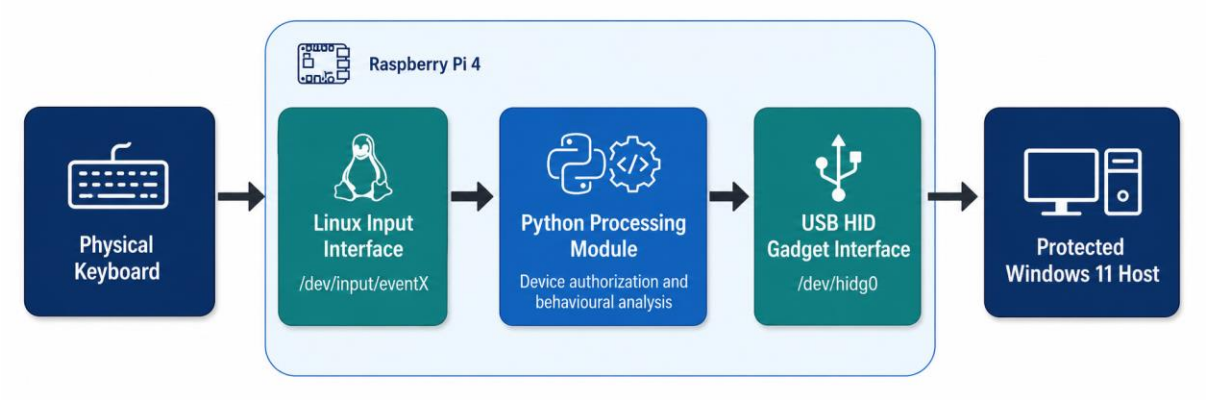

Figure 3.2: Input path from the physical keyboard through the Raspberry Pi to the protected host

Figure 3.2 illustrates the technical path followed by keyboard input. Keystrokes generated by the physical keyboard are captured through the Linux input interface at /dev/input/eventX. The Python processing module performs VID/PID-based device identification, whitelist-based access control, and keystroke-behaviour analysis. Input that satisfies the security checks is converted into valid HID reports and written to /dev/hidg0 through the USB HID gadget interface. The Raspberry Pi subsequently forwards the accepted input to the protected Windows 11 host while preventing unauthorized input from reaching the computer.

Layered Security Controls

The proposed system incorporates four complementary security-control layers. The first layer provides VID/PID-based device identification and whitelist-based access control using records of previously classified devices. Whitelisted devices are allowed to proceed to behavioral monitoring, whereas blacklisted devices are rejected. The second layer analyses keystroke timing to identify potentially automated input. The prototype determines whether the interval between successive keystrokes is unusually short or excessively uniform. The implemented configuration used a fast-typing threshold of 0.03 seconds, an interval-uniformity tolerance of 0.005 seconds, and a suspicious-activity threshold of ten events. The third layer uses CAPTCHA verification to confirm that the connected device is being operated by a human user. CAPTCHA verification is required when an unknown device is enrolled and when the accumulated suspicious-activity count reaches ten events.

Successful verification allows monitoring to continue. If verification repeatedly fails and the maximum number of attempts is exceeded, the device is blocked and disconnected. The fourth layer supports audit and review through separate keystroke and suspicious-event logs. These records allow administrators to review system activity and conduct basic post-event analysis.

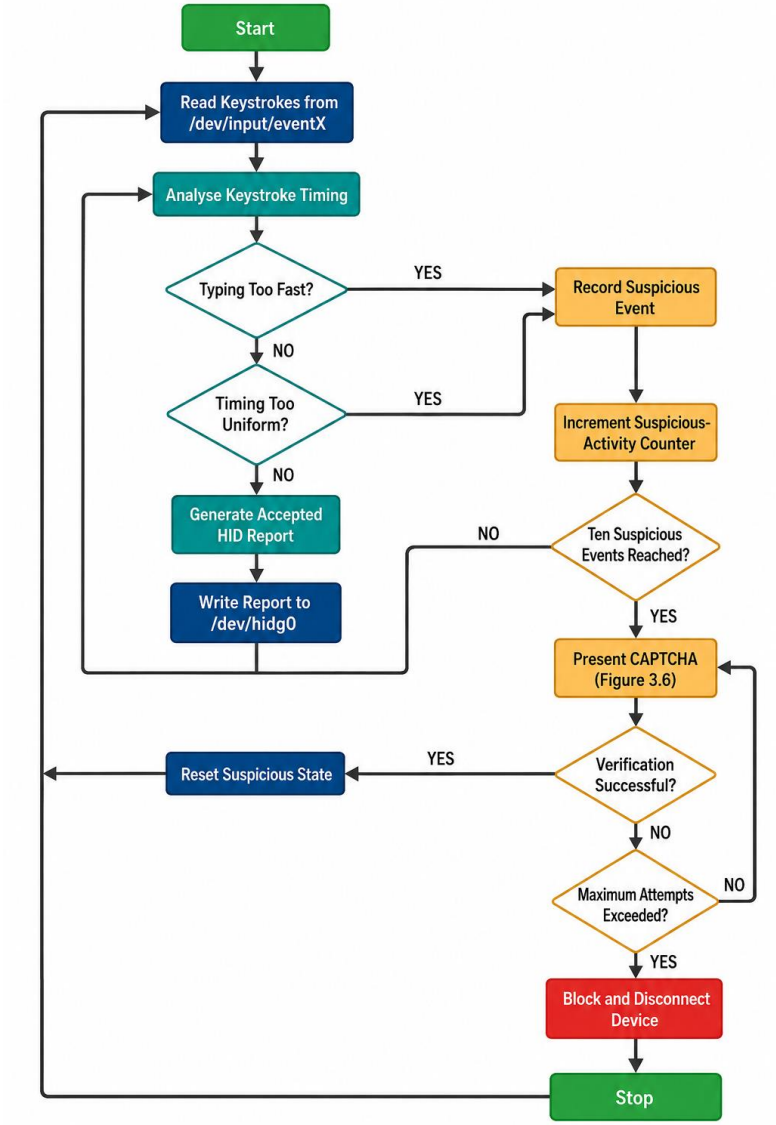

Figure 3.3: Keystroke behavioral analysis and CAPTCHA-triggering workflow

Figure 3.3 illustrates the keystroke behavioural analysis and CAPTCHA-triggering workflow. The system continuously reads keystroke events from /dev/input/eventX and analyses their timing characteristics. It first determines whether the interval between successive keystrokes is shorter than the configured fast-typing threshold. If this condition is not detected, the system evaluates whether the intervals are excessively uniform, which may indicate automated input. When either condition is detected, the event is recorded as suspicious and the suspicious-activity counter is incremented. If the counter remains below ten events, monitoring continues. Once ten suspicious events have accumulated, the system presents the CAPTCHA interface shown in Figure 3.6. Successful verification resets the suspicious state and allows monitoring to continue. If verification fails, the user may retry until the maximum number of attempts is exceeded, after which the device is blocked and disconnected. Input that does not exhibit suspicious timing is converted into a valid HID report, written to /dev/hidg0, and forwarded to the protected host.

Prototype Implementation

The implementation used Python, evdev, Pygame, JSON-based device lists, and Linux USB gadget configuration. The monitoring and graphical-interface functions were separated using threads and event queues

so that input capture, security decisions, and user prompts could operate without blocking one another. Administrative functions included pausing monitoring, viewing or clearing logs, backing up logs, managing the whitelist and blacklist, and terminating the program. Password verification protected administrator-only actions.

Figure 3.4: Normal-use test configuration with the Raspberry Pi 4 positioned between the keyboard and host computer

Figure 3.4 shows the normal-use test configuration of the secure USB passthrough system. The physical keyboard is connected to the Raspberry Pi 4, which functions as an intermediary between the keyboard and the host computer. During operation, the Raspberry Pi captures incoming keystrokes, verifies the connected device, and analyses the keystroke behavior. Input identified as legitimate is converted into HID reports and forwarded to the host computer through the output connection. The integrated display provides information about the system status and detected keyboard activity during testing. This configuration was used to confirm that normal keyboard input could pass through the security mechanism without disrupting regular user interaction.

Figure 3.5: Offensive test configuration using a Raspberry Pi Pico as the malicious HID

Figure 3.5 presents the offensive test configuration used to evaluate the system's ability to detect automated keystroke-injection attacks. The Raspberry Pi Pico was configured to emulate a malicious USB HID device and generate scripted keystrokes at abnormal speeds and highly uniform time intervals. These inputs were transmitted to the Raspberry Pi 4-based secure passthrough system, which analysed the device identification and access-control status together with the keystroke timing before allowing the input to reach the protected host. This configuration was used to determine whether the system could identify suspicious HID behaviour, trigger

CAPTCHA verification after the configured activity threshold, and block or disconnect the device following repeated verification failures.

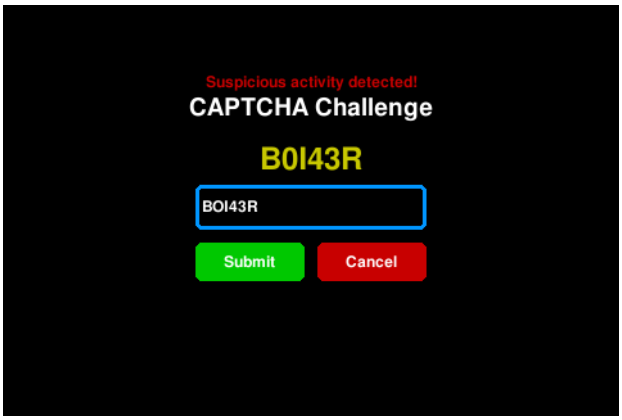

Figure 3.6: CAPTCHA interface used for device enrolment and suspicious-activity verification

Figure 3.6 shows the CAPTCHA interface used to verify that input is being provided by a legitimate human user. When an unknown USB HID device is connected, the system presents the CAPTCHA challenge before allowing the device to be enrolled in the whitelist. The interface is also activated when the accumulated suspicious-activity count reaches the configured threshold of ten events. A successful response permits the device to continue through the enrolment and access-control process, whereas repeated verification failures cause the device to be rejected, blocked, or disconnected from the protected host.

Test Environment and Evaluation Design

The physical test environment consisted of a Royal Kludge H61 keyboard, a Raspberry Pi Pico configured as a BadUSB test device using CircuitPython, a Raspberry Pi 4 Model B running the passthrough application, and an ASUS Zephyrus G15 laptop running Windows 11 Home as the protected host. USB-A-to-USB-C and USB-C-to-USB-C cables provided the required physical connections.

Table 3.1: Test environment

Component	**Platform**	**Role**
Legitimate keyboard	Royal Kludge H61	Normal input source
Malicious HID	Raspberry Pi Pico / CircuitPython	Automated keystroke-injection source
Security intermediary	Raspberry Pi 4 Model B / Raspberry Pi OS	Device identification, access control, behavioural monitoring, verification, and passthrough
Protected host	ASUS Zephyrus G15 / Windows 11 Home	Target receiving accepted HID reports

Attack-driven testing was used for the security functions, while black-box testing evaluated the prototype from the user's perspective. Seven test cases covered device enrolment, normal keyboard operation, reconnection and resilience, malicious-HID detection, CAPTCHA recovery, event logging, and administrative features. A test outcome was marked as passed when the observed behavior matched the specified expected result. These tests establish functional feasibility; they do not constitute a large-sample classification experiment.

Table 3.2: Evaluation test cases

Case	Test Category	Evaluation Purpose
1	Device enrolment	Verify the detection and secure enrolment of an untrusted keyboard
2	Normal operation	Verify standard keystrokes, modifier combinations, and acceptable responsiveness
3	Resilience	Evaluate system behaviour during rapid input, disconnection, and reconnection
4	Security detection	Detect automated BadUSB keystroke injection and trigger CAPTCHA verification
5	Security recovery	Validate CAPTCHA recovery, retry limits, and configured cooldown behaviour
6	Event logging	Verify that relevant suspicious activities are recorded accurately
7	Administrative functions	Validate monitoring controls, log management, device-list management, and program termination

The evaluation was limited to verifying whether the implemented prototype functions operated according to their predefined requirements. It was not designed as a large-sample detection or classification experiment. Each test case comprised one or more observable functional outcomes, producing a total of 18 expected outcomes across the seven test cases. Therefore, these test cases and functional outcomes should not be interpreted as independent legitimate and malicious classification trials. The available data do not support the construction of a statistically meaningful confusion matrix or the calculation of precision, recall, F1-score, or false-positive rate.

Input latency was evaluated under two configurations: a direct keyboard-to-host connection and a connection through the Raspberry Pi-based secure passthrough system. The same keyboard, protected host, and measurement procedure were used in both configurations to maintain consistency. The recorded measurements were averaged for each configuration, and the difference between the two average values was calculated to estimate the additional latency introduced by the secure passthrough. This measurement was treated as a preliminary responsiveness evaluation rather than a comprehensive performance benchmark.

RESULTS AND ANALYSIS

This section presents and analyses the results obtained from the functional testing, normal-use compatibility assessment, and preliminary input-latency evaluation of the secure USB passthrough prototype.

Functional Test Results

The prototype was evaluated against 18 predefined functional outcomes across seven test cases. Table 4.1 summarizes the number and percentage of outcomes that passed, partially passed, or failed.

Table 4.1: Summary of expected functional outcomes

Outcome	Count	Percentage	Interpretation
Passed	16	88.9%	Observed behavior matched the expected result
Partially passed	1	5.6%	Standard OS shortcuts worked, but one keyboard-specific Fn combination was not forwarded correctly.

Failed	1	5.6%	Rapid legitimate input triggered the suspicious-activity threshold, resulting in a false-positive event.
Total	18	100.0%	All expected outcomes across seven test cases

The prototype completed most of the predefined functional outcomes. Of the 18 outcomes evaluated, 16 passed, one partially passed, and one failed, producing a functional pass proportion of 88.9%. This percentage represents the completion of predefined functional checks and is not a measure of attack-detection accuracy. The functional test design did not generate the repeated labelled legitimate and malicious observations required to determine true positives, false positives, true negatives, or false negatives. Consequently, the results cannot be used to calculate a statistically valid confusion matrix, precision, recall, F1-score, or false-positive rate.

Normal-Use Compatibility

Ordinary alphanumeric input and operating-system-level shortcuts, including Ctrl+C and Win+I, were forwarded successfully. A keyboard-specific Fn+F11 volume-control combination did not operate correctly because the function-key behavior depended on the keyboard's own firmware or driver rather than only on the standard raw HID report handled by the passthrough. This result indicates that broader keyboard compatibility cannot be assumed from successful testing with standard key codes. Rapid legitimate interaction occasionally generated a false alarm under the configured timing thresholds. The CAPTCHA allowed the user to demonstrate human presence and resume normal operation, preventing immediate permanent blocking. Nevertheless, the event remained a false positive. The result confirms the need to calibrate behavioral thresholds using a larger and more diverse set of human typing patterns.

Input Latency

The reported average input latency was 96.72 ms when the keyboard was connected through the secure passthrough and 95.57 ms when connected directly to the host. The measured difference was therefore 1.15 ms. Under the tested configuration, the observed difference provides a preliminary indication that the passthrough introduced only a small additional delay. However, this result should not be interpreted as a statistically validated or generalisable latency measurement. Future testing should use a documented measurement procedure and report the sample size, variance, standard deviation, confidence intervals, and detection latency.

Table 4.2: Preliminary input-latency comparison

Connection Configuration	**Average Latency (ms)**	**Additional Latency (ms)**
Direct keyboard-to-host connection	95.57	Baseline
Secure Raspberry Pi passthrough	96.72	+1.15 ms

DISCUSSION

The results support the feasibility of using a Raspberry Pi as a hardware-mediated security boundary for HID input. The main architectural advantage is separation: the peripheral communicates with the Raspberry Pi rather than directly with the protected computer. This arrangement allows security decisions to be made before input reports are forwarded and reduces dependence on host-side antivirus or local device-control configuration. The prototype also demonstrates the value of combining controls with different purposes. VID/PID-based identification recognises the reported device identifiers, whitelisting provides access control for known devices, behavioural timing rules identify conspicuously automated input, CAPTCHA provides interactive human verification after uncertainty, and logging preserves events for subsequent review. Although no individual layer provides conclusive protection, their integration establishes a practical layered defence. The prototype is also comparatively inexpensive: the reported hardware configuration used a Raspberry Pi 4, Raspberry Pi Pico,

cables, and storage with an estimated total cost below RM400 while relying primarily on open-source software components.

However, the current evidence should be interpreted conservatively because this study evaluated implementation functionality rather than general detection performance. The experiment used one legitimate keyboard, one host platform, one Raspberry Pi Pico attack device, and a limited set of predefined scenarios. It did not include multiple legitimate keyboards, users, host operating systems, BadUSB devices, injection speeds, randomised delays, human-like timing patterns, VID/PID-spoofing conditions, composite USB devices, or additional HID classes. These factors remain outside the scope of the present proof-of-concept evaluation. Moreover, the VID/PID mechanism should be understood as device identification and whitelist-based access control rather than cryptographically secure device authentication. The false-positive event observed during rapid legitimate input further indicates that fixed behavioural thresholds may not generalise across different users and typing patterns.

A future detection-performance study should use repeated labelled legitimate and malicious trials to construct a formal confusion matrix and calculate precision, recall, F1-score, and false-positive rate. Testing should involve multiple users, legitimate keyboards, host operating systems, BadUSB devices, injection speeds, randomised delays, human-like attack patterns, and VID/PID-spoofing scenarios. Future work should also evaluate adaptive or user-specific behavioural thresholds using larger legitimate-typing datasets, strengthen device authentication beyond VID/PID-based identification, and assess compatibility with composite USB devices and additional HID classes. Latency testing should employ a clearly documented measurement procedure and report the sample size, variance, standard deviation, confidence intervals, and detection latency.

CONCLUSION

This study developed a Raspberry Pi-based secure USB passthrough system for mitigating HID-based BadUSB keystroke-injection attacks. The prototype combined device identification, whitelist and blacklist controls, real-time keystroke-timing analysis, CAPTCHA verification, input forwarding, logging, and administrative functions in a portable intermediary device. Sixteen of the 18 predefined functional outcomes passed, one partially passed, and one failed. The tested automated injection was detected and interrupted, while the secure passthrough introduced an observed average latency difference of 1.15 ms relative to the direct connection. However, this latency finding represents a preliminary observation rather than a statistically validated performance result. The occurrence of a false-positive event during rapid legitimate input demonstrates that CAPTCHA functions as a recovery mechanism rather than evidence of accurate attack classification. The results establish the preliminary functional feasibility of the prototype under the tested configuration. They do not establish general detection accuracy, cross-platform compatibility, or resistance to sophisticated human-like attacks. A larger experimental study involving repeated legitimate and malicious trials is required to evaluate classification performance, adaptive behavioural thresholds, stronger device-authentication mechanisms, compatibility with composite USB devices, and support for additional HID classes.

ACKNOWLEDGEMENT

The authors would like to thank Universiti Teknikal Malaysia Melaka (UTeM) and Universiti Teknologi MARA (UiTM) Jasin Campus for the support.

REFERENCES

1. Axelson, J. (2015). USB complete: The developer’s guide. Lakeview Research.
2. Fakiha, B. S. (2024). Forensic analysis of BadUSB attacks: A methodology for detecting and mitigating malicious USB device activities. Edelweiss Applied Science and Technology, 8(5), 1090–1100.
3. Charan, V., & Kulkarni, L. (2023). Survey on micro-controller-based BadUSB attacks. Journal of Positive School Psychology, 7(1), 965–974.
4. Nissim, N., Yahalom, R., & Elovici, Y. (2017). USB-based attacks. Computers & Security, 70, 675–688.
5. Karystinos, E., Andreatos, A., & Douligeris, C. (2019). Spyduino: Arduino as a HID exploiting the BadUSB vulnerability. In Proceedings of the IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS).

6. Nicho, M., & Sabry, I. (2023). Bypassing multiple security layers using malicious USB Human Interface Device. In Proceedings of the 9th International Conference on Information Systems Security and Privacy (pp. 501–508).
7. Tian, D. J., Bates, A., & Butler, K. (2015). Defending against malicious USB firmware with GoodUSB. In Proceedings of the 31st Annual Computer Security Applications Conference.
8. Wang, C.-Y., & Hsu, F.-H. (2024). USBIPS framework: Protecting hosts from malicious USB peripherals. arXiv.
9. Dumitru, R., Beaumont, M., & Hopkins, B. (2023). USB proxy. In Proceedings of the 2023 Australasian Computer Science Week (pp. 122–125).
10. Koffi, K. A., Smiliotopoulos, C., Kolias, C., & Kambourakis, G. (2024). To (US)Be or not to (US)Be: Discovering malicious USB peripherals through neural network-driven power analysis. Electronics, 13(11), Article 2117.
11. Seo, J.-H., & Moon, J.-S. (2017). Analysis and countermeasure for BadUSB vulnerability. IEMEK Journal of Embedded Systems and Applications, 12(6), 359–368.
12. Blanchet, S. (2018). BadUSB: The threat hidden in ordinary objects. ResearchGate.
13. Pham, D. V., Syed, A., & Halgamuge, M. N. (2011). Universal Serial Bus-based software attacks and protection solutions. Digital Investigation, 7(3–4), 172–184.
14. Dasgupta, R. K. (2018). Turning regular AVR microcontroller to Human Interface Device (HID) for hacking and penetration testing.