Screen Readers – Out of Sight, but in the TCB

Sebastian Humenda, Samuel Thibault, Horst Schirmeier

To cite this version:

Sebastian Humenda, Samuel Thibault, Horst Schirmeier. Screen Readers – Out of Sight, but in the TCB. 2025 - Herbsttreffen der Fachgruppe Betriebssysteme, Sep 2025, Aachen, Germany. ⟨10.18420/fgbs2025h-03⟩. ⟨hal-05273382⟩

HAL Id: hal-05273382
https://hal.science/hal-05273382v1

Submitted on 22 Sep 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Screen Readers – Out of Sight, but in the TCB

Sebastian Humenda
sebastian.humenda@secunet.com
secunet Security Networks AG
Dresden, Germany

Samuel Thibault
samuel.thibault@u-bordeaux.fr
University of Bordeaux
Talence, France

Horst Schirmeier
horst.schirmeier@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

Abstract

Modern operating systems (OSs) must balance strong security guarantees with accessibility obligations for blind users mandated by regulation and principles of equal treatment. Screen readers (SRs) translate user interfaces into braille or speech, but today's SRs operate as large, monolithic processes with privileged access to input, output, and application state. This design places SRs in the trusted computing base (TCB) and exposes blind users to high risks: A compromised SR can inject input, capture sensitive output, or impersonate trusted system messages.

We argue that SR functionality should be refactored into OS services with strict isolation and least privilege. Our proposed architecture decomposes input handling, output multiplexing, and braille-device access into components mediated by the window system and the kernel. This separation removes global event channels, reduces the TCB, and limits escalation paths. Our design aligns SR security with existing OS protection mechanisms, thereby shrinking the attack surface for a critical yet under-protected user group.

1 Introduction

General-purpose computing devices such as laptops, desktop PCs, and mobile phones are critical for everyday life. Users rely on them for communication, banking, shopping, and storing personal data. We call a system *trusted* if it enforces access control and isolation, and provides trusted input/output to preserve data integrity, privacy and confidentiality. A large TCB increases complexity and the attack surface [20]. Thus, the goal is to minimise trusted components even in consumer systems. Furthermore, it is crucial to apply the principle of least privilege to trusted components.

Regulatory obligations [7, 17] and general principles of equal treatment mandate accessibility of systems for blind users. To meet these obligations, OS vendors must provide screen readers (SRs). SRs provide access to an alternate user-interface (UI) representation using braille and speech. They receive events describing the UI layout and changes by a dedicated platform-specific *accessibility API*, but also inject events like mouse clicks. We focus on braille output for blind users, as covering all disabilities and modalities comes with in part contradicting trade-offs and constraints on the design and are beyond the scope of this paper.

System components handling user interaction are critical for security as they process untrusted data and may pierce isolation domains. SRs fall into this category: They capture input, intercept keystrokes, drive braille devices, broker I/O, and render UI feedback [13, 19]. Figure 1 shows two browser windows run in separate processes (shopping and banking), yet the SR can read events from both using the accessibility API. It can also inject keystrokes into the unfocused application, enabling credential theft or unauthorised actions. The monolithic architecture and excessive privileges conflict with security objectives for user-data privacy, integrity, and confidentiality. This design probably results from OS vendors adding accessibility support as an afterthought: Vendors added accessible output as a layer onto existing systems, which trades off between security and compatibility [14].

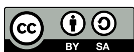
For Fig. 1, we assume that the attacker exploited the SR, using one of several attack vectors resulting from the large and complex TCB, for instance, by a bug in the braille text translation [3]. First, the attacker can circumvent the process isolation employed by modern browsers and inject key presses into the window with the logged-in bank account. Second, an attacker is able to snoop the credentials from the user during banking.

The contributions of this article are as follows:

- On the example of GNU/Linux, we discuss general SR-architecture deficiencies that can undermine privacy, data integrity, and confidentiality.
- We define a threat model illustrating the weaknesses of current SR architectures.
- We propose a new SR architecture that embeds the SR into core OS services to leverage existing security mechanisms.

2 Background and State of the Art

Understanding the internal structure of SRs is key to analysing their security impact. This section explains how SRs operate at the system level, using GNU/Linux as a case study. SRs on other platforms differ in implementation but share the same monolithic architecture co-locating a wide set of functionalities: intercepting keystrokes, driving braille devices,



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. DOI: https://dx.doi.org/10.18420/fgbs2025h-03. GI/ITG FG Operating Systems, September 25–26, 2025, Aachen, Germany.

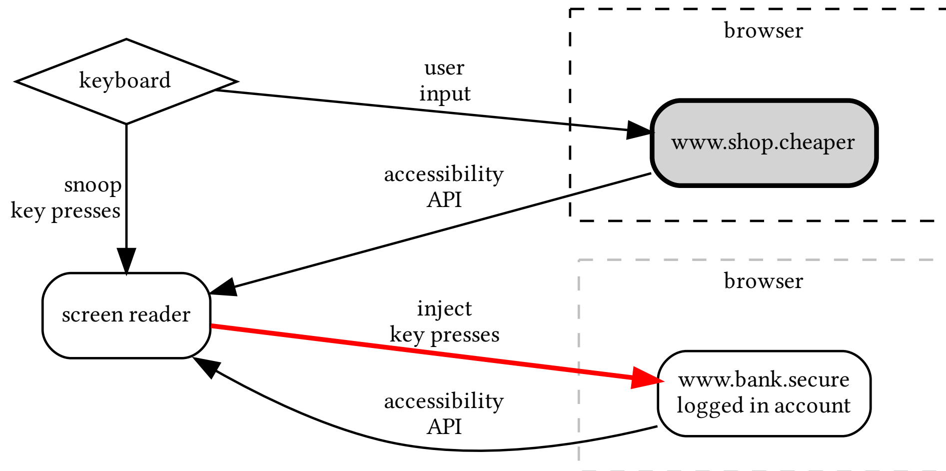


Figure 1: Two browser windows, accessed by an SR. Dashed borders denote the process isolation boundary. The browser window with a shopping site has focus (emphasized border) and receives user input, whereas the SR injects key presses into the bank site while in the background.

brokering I/O, and rendering UI feedback. This enlarges the TCB, as SR processes require a broad set of privileges to operate.

The SR also tracks application focus independently, duplicating the window manager's role [2]. External focus tracking is error-prone and can lead to confusion which application is active [6, 21], opening a potential attack surface.

2.1 Background on Braille Input/Output

A refreshable braille display is an electro-mechanical device to display text as braille characters. Because of physical size limits and cost, most braille displays expose a single line of 40–80 characters. To compensate for the small size of a braille display, the SR provides navigation commands for UI overview. The SR implements these by a combination of braille display keys and keyboard commands, for which it needs to snoop all keyboard inputs. For a subset commands, it injects synthetic input events into the OS, for example to simulate mouse clicks.

2.2 AT-SPI on GNU/Linux

On GNU/Linux desktops, the central accessibility framework is the *Assistive Technology Service Provider Interface* (AT-SPI). AT-SPI defines how applications expose UI state and events, and how assistive tools query or control these events. AT-SPI uses the D-Bus message bus, making accessibility events available to all processes in a user session.

On GNU/Linux, braille drivers are separated from the GUI SR, Orca, shown in Figure 2. BRLTTY is a command-line SR with a wide range of drivers that reflect the fragmented braille display landscape. Through `BrlAPI` [19], graphical SRs such as Orca can reuse these drivers. BRLTTY is complex: Version 6.8 comprises ~156 000 lines of code[1], excluding speech drivers and support for Windows and Android. In default setups, BRLTTY runs as a privileged system service, so any driver compromise could escalate to full system access.

Figure 2 illustrates the central role of the AT-SPI stack on an GNU/Linux desktop. Applications send events to the per-user bus. The SR *Orca* subscribes to these events, filters those of interest, and constructs an internal object tree representing the accessible UI. Then it renders the object tree to a representation for the braille display and sends it to BRLTTY. This model enables Orca to present structured content on a braille display and to convert braille key presses into events injected into applications. Because these steps run inside one privileged process, any compromise exposes the entire input and output path, growing the TCB.

Another weakness of this design is the shared per-user accessibility bus, which allows Orca to simultaneously receive events from all applications. Using again the example of Fig. 1, this design violates confidentiality: It creates an information flow between two otherwise isolated processes – apart from the window manager (WM), which shares this problem to a certain extent. Furthermore, on X.Org desktops, an application can even send input events to arbitrary applications, including those in the background. X.Org is still widely in use by blind users because Wayland lags behind

[1]Effective lines of code (excluding empty lines and comments), obtained with *cloc* [5].

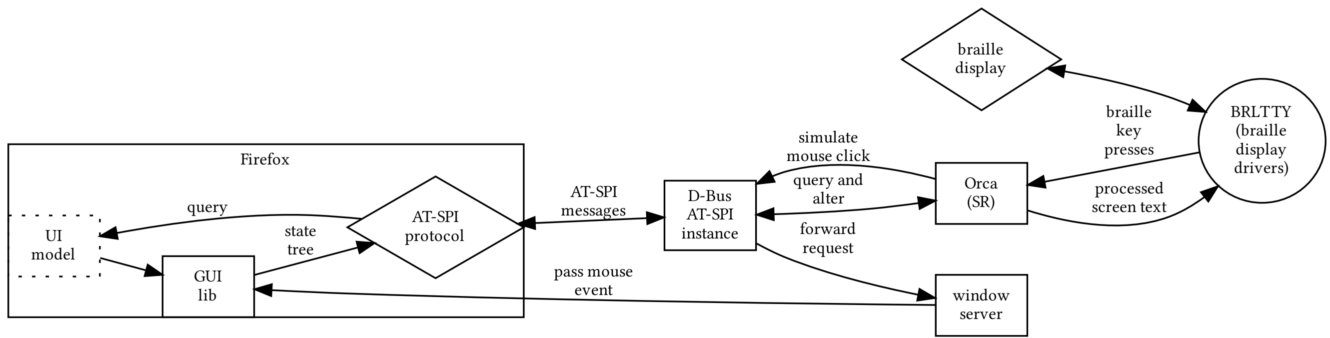


Figure 2: Overview of the GNU/Linux accessibility architecture on X.Org desktops. Applications communicate with AT-SPI by D-Bus. Orca subscribes to events, maintains a UI model, and can query or inject events. BRLTTY functions as a braille-driver reservoir.

in accessibility support. This allows an attacker to carry out hidden actions.

2.3 Contrasting Accessibility and OS Abstractions

From a systems perspective, AT-SPI duplicates core OS abstractions without the same privilege separation and security hardening.

Input/Output Multiplexing. In conventional OS design, kernel drivers handle raw input while window servers mediate output, enforce isolation, and compose the screen. SRs break this separation: They combine braille drivers, input injection, and output multiplexing, tracking focus and rendering application accessibility output in one process. On GNU/Linux, braille drivers run separately in BRLTTY yet still with root privileges. Orca bundles input handling and multiplexing, preserving a monolithic design and duplicating OS/WM functionality. Neither X.Org nor Wayland natively support braille as an output channel, leaving this functionality in a privileged user-space process outside of OS mediation.

Privilege Separation and Trust Signalling. All widely available accessibility frameworks lack explicit privilege transitions and unforgeable provenance indicators. As a result, braille output remains vulnerable to replay or spoofing. Research prototypes such as Nitpicker [8, 9] proposed a minimal authoritative multiplexer for inputs and graphical outputs with user-controlled focus. An important property was the labelling of application windows with their respective trust level by the WM. Hence, applications could not tamper with the trust labelling. The same principles apply to braille multiplexing: A single minimal service should control global focus and labelling, while a separate policy component enforces which processes may emit braille output.

2.4 Related Work

Recent studies highlight security risks in accessibility frameworks across platforms [10, 13, 14, 15, 16, 18]. Zezulak et al. [22] claim that for web applications, the overlap of accessibility and security needs more research focus. We argue that desktop-OS architectures suffer from the same deficiency, and work towards providing a remedy. Similarly, Renaud and Coles-Kemp [18] identify security, usability, and accessibility as relevant dimensions of systems design and warn that designs neglecting accessibility exclude users and increase exposure to attacks.

Jang et al. [14] perform the first cross-platform analysis of accessibility APIs on Windows, Linux, iOS, and Android, revealing twelve attack vectors from unchecked UI automation channels, including user-account control (UAC) bypass and control over input events. They stress that accessibility paths often omit essential security checks due to compatibility concerns, but do not propose architectural redesigns.

For Android, Kalysch et al. [15] present data leaks in popular apps by abused accessibility services, and propose to harden applications against these weaknesses. Naseri et al. [16] confirm this at scale for finance and social-media apps. They identify the granularity of the accessibility permission as too coarse, proposing a split into smaller permissions. Both works [15, 16] focus on the permissions of the SR and do not address the conflation of input event handling, input injection, and output rendering on Android. Fratantonio et al. [10] show how extensive accessibility and overlay permissions enable stealthy key logging and full UI control. They propose changes to the UI framework to mark applications or individual UI widgets as secure that would change the interaction with the SR. This proposal does not alter the general accessibility design of Android.

Huang et al. [13] propose a modular Android architecture that separates UI analysis, input mediation, and output

handling into sandboxed services with least-privilege enforcement. While related, our approach integrates SR functionality into the OS itself rather than isolating parts of a monolithic service.

3 Threat Model

From the weaknesses of existing architectures, we can derive requirements for secure systems with an SR. A secure design requires a clear statement of the assets an attacker can attack, and a description of the setting of such an attack. We use the threat model to describe the capabilities of an attacker and to define attacks that are in scope of this paper. We define which components we consider to be part of the TCB and conclude with the security objectives of our design.

We target general-purpose OSes with accessibility for blind users, focusing on braille and assuming standard OS isolation mechanisms. Further, we assume a WM that multiplexes input/output solely to the active application.

Setting: A multi-application user session with OS accessibility enabled. Applications expose UI state via an accessibility interface. Braille devices connect over serial/USB/Bluetooth. The SR is either a separate process integrated with OS services or embedded as part of the application toolkit.

Assets: Confidentiality of UI data between focused and non-focused applications, user-input integrity, trustworthy application-focus indicator to the user, process isolation for accessibility components, and controlled access to braille devices.

TCB:
- Kernel, WM, driver sandbox, minimal drivers, and minimal policy services.
- Not in the TCB: applications, device firmware.

Adversary: local apps, remote content, or malicious devices exploiting buggy drivers.

Adversary Capabilities: Inject input events, observe confidential data across applications, exploit protocol flaws, fake application focus, or abuse legacy privileges.

Security objectives:

S.1 No cross-application data flow via accessibility: The SR must not expose non-focused application state or read global input.

S.2 Input integrity: Solely the WM may deliver input to the focused application; accessibility components must not create cross-application injection paths.

S.3 Provenance on braille: The user can rely on unforgeable association between braille output and the focused application.

S.4 Fault isolation: Bugs in braille drivers, `libsr`, or the SR process do not break OS-provided isolation or leak application state.

S.5 Least privilege: Sensitive components require isolation and sandboxing.

Out of scope:
- Kernel compromise, physical side channels, or DMA attacks.
- Speech output channels and audio spoofing.
- Covert channels between colluding applications beyond the accessibility interface.

In this model, the threat driver is the legacy monolithic design: Global event paths, SR-controlled input injection, and in-process drivers enlarge the TCB and enable cross-app flows. The proposed architecture mitigates these by WM-mediated focus, sandboxed drivers, and a reduced SR or a library SR, introduced in the following section.

4 Decentralised Screen Reader Architecture

We propose to overcome limitations of current SR architectures through a careful SR-OS co-design. We identify the following core building blocks for an SR: braille drivers, braille input handling, keyboard input handling, UI tree rendering, braille translation, and output multiplexing. Throughout this section, we separately discuss each building block and its integration into OS services.

Our architecture assumes that the windowing system blocks global input injection and forwards events solely to the focused application (objective S.2). For instance, Wayland enforces this. We leave external focus change events, such as notifications, for future work. Focus stealing is dangerous because the user could accidentally give away secrets to the focus-stealing application, before noticing.

4.1 Screen Reader Building Blocks

Figure 3 shows the proposed decentralized architecture. We describe its building blocks below.

Braille device drivers. We propose moving drivers into a separate, isolated sandbox. The sandbox can access the braille display, yet has no access to the file system, the process tree, and the network (objectives S.4, S.5). Optionally, a proxy can implement network transparency akin to BRLTTY's `BrlAPI` outside the sandbox.

Braille Device Input. Braille input needs to become an input method mediated by the WM, akin to keyboards (objective S.2). We identify two places to implement vendor-specific key mappings: in the driver sandbox or in the SR. The former simplifies the protocol, while the latter gives more flexibility for SR-specific functionality.

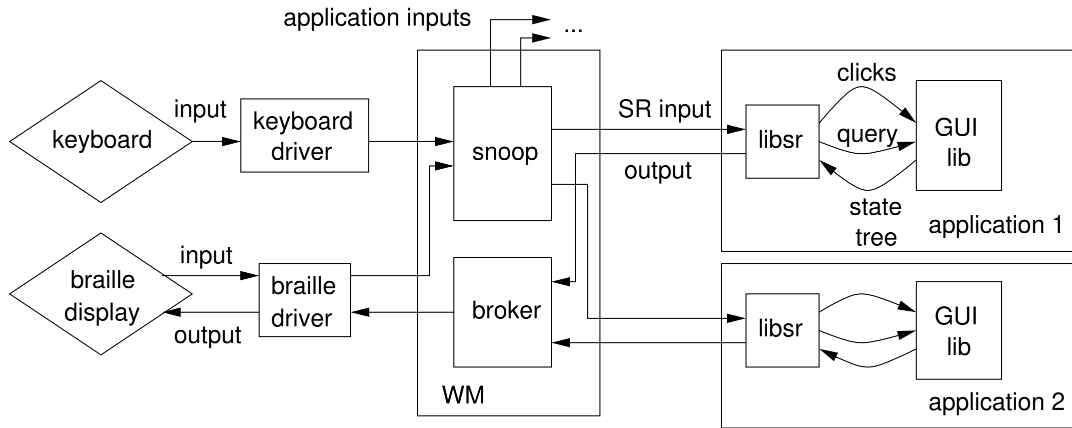


Figure 3: Overview of our proposed architecture with SR as a library. The WM separates out SR input snooping with a modifier key from standard application input, and multiplexes output from SR. `libsr` interacts directly with the application toolkit to query the state tree and inject mouse clicks.

Keyboard-based navigation. To avoid a general key snooping mechanism in the SR, the WM needs to add an API to register so called *modifier keys* (objective S.1). The WM delivers key combinations containing the modifier to the SR. Wayland enforces such a design [4].

Application-output multiplexing and focus tracking. In order to avoid communication of accessibility API messages from background applications, the WM must be in charge of multiplexing these messages to the SR (objective S.1). On focus switch, the WM needs to revoke the communication channel from the current application to the SR and grant the SR access to the newly focused application. As an alternative, the WM can also act as a broker for the output, avoiding frequent resource revocation.

4.2 Screen Reader as a Deprivileged Process

Our architecture preserves compatibility with a process-based SR while integrating all other building blocks into OS services. The WM mediates accessibility API output and focus-change notifications and forwards them to the SR. The SR renders the accessibility tree, translates it into braille, and interacts with the driver sandbox through a well-defined protocol. This design already removes global input channels and arbitrary inter-process communication present in legacy architectures; it also fulfils all security objectives.

4.3 Screen Reader per Application

We can further isolate application state and minimise the leakage of data through the SR by running an SR per application. In case of a compromise of the SR, the attacker remains isolated to accessing a single application. This also simplifies the mediation of the WM, as accessibility messages can directly flow from application to the SR and back. This design reduces the risk of side-channels, as dedicated communication between SR and application avoid a share bus or mediator.

4.4 Screen Reader as a Library

The designs so far preserve a separate SR process, isolated and mediated by the WM. A more radical step removes this process boundary entirely: The SR becomes a library, `libsr`, linked into each application via its UI toolkit. The toolkit augments its UI representation with accessibility metadata or maintains a parallel accessibility tree consumed by `libsr`. Like modern graphics stacks where rendering executes in the application itself [12, 23], `libsr` renders all accessibility output locally.

The WM enforces focus and mediation policies. When the application has focus, the WM retrieves the braille buffer from `libsr` and forwards it to the driver sandbox. The WM also communicates device properties such as cell dimensions and refresh rates, similar to how graphical parameters reach rendering libraries. This arrangement confines accessibility data and output generation to the application boundary until the WM explicitly authorises output to the device.

Integrating `libsr` into toolkits reduces complexity in three ways:

(1) It eliminates legacy accessibility APIs and their associated marshalling overhead (objective S.1).
(2) It removes the need for separate SR processes in most scenarios; output occurs in the application context.
(3) It avoids global event channels: Solely the WM mediates focus changes and device access.

Another positive effect is a potential increase in SR performance, as library calls require no context switches or IPC. The downside of the `libsr` variant is that in contrast to the other proposals, `libsr` puts the application at risk, as it adds more code into the application that is potentially vulnerable. As per the threat model, the application is not part of the TCB; `libsr` would need a different threat model, which we leave for future work. For security-sensitive systems, `libsr` still a valid choice, as it eliminates side channels.

4.5 Deployment Considerations

Migration towards `libsr` can proceed incrementally: Modern toolkits adopt `libsr` natively, while legacy applications rely on compatibility mechanisms such as `LD_PRELOAD` or DLL injection like it is state of the art for accessibility on Windows systems [1, 11]. This preserves interoperability without weakening the isolation guarantees of the new architecture.

All three SR deployment variants share the same security primitives: sandboxed drivers, WM-mediated input, and strict output multiplexing. This takes the SR out of the critical path. The library SR is preferable, as aligns accessibility output with established rendering paradigms and minimises trusted code outside the OS and toolkit. The SR-per-application approach trades duplication for increased isolation and does not require adapting applications. It is hence preferable in commodity systems.

Trust Indicator. For braille output, we propose to use a label to display the current trust level. The WM implementation can choose the label and its use, for instance, to show administrative privileges. Solely the WM sets the label in a tamper-proof manner. We propose two options:

(1) Reservation of a cell of the display to show the current trust label of the message. The WM decreases the cell count of the display passed to the SR/`libsr` so that the application cannot forge the label.
(2) Introduction of a *secure attention key* (SAK), that is, a keyboard key combination that restricts output to the WM while held. While holding the SAK, the user could query the title of the application or its trust label.

The advantage of the reserved braille cell is its conceptional simplicity, while it reduces the already limited amount of space on a braille display. The SAK allows for more flexibility, but remains tied to systems with a keyboard.

5 Conclusion and Future Work

We examined security risks in current SR architectures, focusing on braille output as a representative accessibility channel. Our analysis shows that monolithic SR designs enlarge the TCB and enable cross-application attacks when integrated without strict OS mediation. We proposed a refactored architecture that embeds SR functionality into core OS services, enforces isolation between SR building blocks and applications, and applies least-privilege principles to drivers and multiplexing.

Future work includes implementing and evaluating this architecture on a modern OS platform to assess performance, compatibility, and security trade-offs. One open question is whether a global UI view is necessary in certain circumstances, for instance for rendering notifications or events spanning multiple applications. We also plan to study whether the restriction to a single application poses problems for the SR when multiple sources compete for attention, e.g. an incoming call on a phone. Formal verification of isolation properties and support for additional output channels, such as speech, remain further research directions.

References

[1] Le An, Marco Castelluccio and Foutse Khomh. 2019. An empirical study of DLL injection bugs in the Firefox ecosystem. *Empirical Software Engineering*, 24, 4, 1799–1822. doi:10.1007/s10664-018-9677-7.

[2] 2025. AT-SPI StateType enumeration (assistive technology SPI states). GNOME Project. Retrieved 15th Sept. 2025 from https://docs.gtk.org/atspi2/enum.StateType.html.

[3] Hongxu Chen. 2018. CVE-201817294. Stack-buffer-overflow in Liblouis 3.6. (Sept. 2018). Retrieved 1st June 2025 from https://www.cve.org/CVERecord?id=CVE-2018-17294.

[4] Jonathan Corbet. 2025. Enhancing screen-reader functionality in modern GNOME. *LWN.net*, (17th June 2025). Retrieved 20th Aug. 2025 from https://lwn.net/Articles/1025127/.

[5] [SW] Albert Danial, cloc: Count Lines of Code version v1.92, Dec. 2021. doi:10.5281/zenodo.5760077.

[6] Joanmarie Diggs. 2023. GNOME/orca issue #434. [KEYHANDLING] [i3] a copious amount of window activation/deactivation events happen in (at least) Firefox, messing Orca up. GNOME Project. Retrieved 15th Sept. 2025 from https://gitlab.gnome.org/GNOME/orca/-/issues/434.

[7] 2019. Directive (EU) 2019/882 of the European Parliament and of the Council of 17 april 2019 on the accessibility requirements for products and services. OJ L 151, 07.06.2019, pp. 70–115. Directive. European Union, (17th Apr. 2019). Retrieved 25th Feb. 2024 from https://data.europa.eu/eli/dir/2019/882/oj.

[8] Norman Feske. 2009. *Securing Graphical User Interfaces*. PhD thesis. Technische Universität Dresden.

[9] Norman Feske and Christian Helmuth. 2005. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 85–94. doi:10.1109/CSAC.2005.7.

[10] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung and Wenke Lee. 2017. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1041–1057. doi:10.1109/SP.2017.39.

[11] Neill Hadder. 2023. Your browser may be having a secret relationship with a screen reader. Knowbility Inc. (July 2023). Retrieved 15th Sept. 2025 from https://knowbility.org/blog/2023/accessibility-apis-part-3.

[12] Kristian Høgsberg. 2012. Wayland protocol documentation – introduction. Retrieved 13th Sept. 2025 from https://wayland.freedesktop.org/docs/html/ch01.html.

[13] Jie Huang, Michael Backes and Sven Bugiel. 2021. A11y and privacy don't have to be mutually exclusive: constraining accessibility service misuse on Android. In *30th USENIX Security Symposium*, 3631–3648.

[14] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang and Wenke Lee. 2014. A11y attacks: exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 103–115. doi:10.1145/2660267.2660295.

[15] Anatoli Kalysch, Davide Bove and Tilo Müller. 2018. How Android's UI security is undermined by accessibility. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, 1–10.

[16] Mohammad Naseri, Nataniel P. Borges Jr., Andreas Zeller and Romain Rouvoy. 2019. AccessiLeaks: investigating privacy leaks exposed by the Android accessibility service. In *The 19th Privacy Enhancing Technologies Symposium (PETS 2019)*.

[17] 1998. Rehabilitation act of 1973, § 508 (as amended 1998). U.S. Code. Accessibility requirements introduced by the 1998 amendment. (1998). Retrieved 26th Apr. 2025 from https://www.section508.gov/manage/laws-and-policies/.

[18] Karen Renaud and Lizzie Coles-Kemp. 2022. Accessible and inclusive cyber security: a nuanced and complex challenge. *SN Computer Science*, 3, 5, 346. doi:10.1007/s42979-022-01239-1.

[19] Samuel Thibault and Sébastien Hinderer. 2007. BrlAPI: simple, portable, concurrent, application-level control of braille terminals. In *The First International Conference on Information and Communication Technology and Accessibility – ICTA 2007*. Hammamet, Tunisia, (Apr. 2007), 27–31. https://inria.hal.science/inria-00135946.

[20] U.S. Department of Defense. 1985. Trusted Computer System Evaluation Criteria. DoD 5200.28-STD (Orange Book). National Computer Security Center.

[21] Colomban Wendling. 2019. GNOME/orca issue #29. Lack of window:activate event leads to not presenting object. GNOME Project. Retrieved 15th Sept. 2025 from https://gitlab.gnome.org/GNOME/orca/-/issues/29.

[22] Alisa Zezulak, Faiza Tazi and Sanchari Das. 2023. SoK: evaluating privacy and security concerns of using web services for the disabled population. In *Proceedings of the 7th Workshop on Technology and Consumer Protection (ConPro '23)*. (May 2023).

[23] Thomas Zimmermann. 2023. The Linux graphics stack in a nutshell, part 1. *LWN.net*, (19th Dec. 2023). Retrieved 13th Sept. 2025 from https://lwn.net/Articles/955376/.