ChatGPT

NavSentinel: Enhancing Navigation Intent Filtering

Context and Existing Solutions

NavSentinel's approach addresses a well-known gap in browser defenses: many "pop-under" or deceptive navigations bypass built-in popup blockers by piggybacking on real user clicks [1]. Research shows these malicious behaviors (e.g. overlay clickjacking, hidden redirect chains) are **extremely common**, even on popular sites [2] [3]. In fact, nearly all new-tab navigations opened via HTML or JavaScript suffer from insecure or unexpected behavior, yet defenses (like `noopener` or strict blockers) are **rarely implemented by sites** [4].

Existing extensions take crude approaches. For example, *No New Tabs* (formerly "Death To _blank") simply forces every link or script-opened page into the same tab, blocking new tabs entirely [5]. The latest version had to get "unbreakable" by intercepting **every technique** sites use to spawn tabs, beyond just stripping `target="_blank"` [5]. Another tool, *Popup Blocker (strict)*, **intercepts all new window requests** (from `window.open`, target=_blank links, even form submissions) and pauses them for user approval [6] [7]. These tools prove it's possible to catch popups early, but they treat **all new tabs as bad by default**, relying on whitelists or manual user decisions [8] [9]. NavSentinel aims to be smarter: using a risk scoring model to distinguish legitimate user-intended navigations from malicious ones, thereby minimizing needless prompts. This aligns with academic proposals – researchers even built a proof-of-concept extension to prevent "dangerous side effects" from clicks [10] – but NavSentinel intends to push this concept further with a transparent, gesture-based intent verification system.

Architecture Review and Suggested Enhancements

Gesture Token & Scoring Mechanism: NavSentinel's core idea is to issue a short-lived **GestureToken** on every genuine user click (pointerdown) and then require any new navigation to present a matching token. This is a sound strategy to ensure the navigation is truly user-initiated. An improvement here is to make the token carry context: e.g. include the coordinates and element signature of the click. This way, when a navigation attempt occurs, NavSentinel can compare it against the original click target. The proposed **Click Deception Score (CDS)** features – checking if the clicked element was an overlay, if it changed between pointerdown and click, etc. – are excellent. We can enhance this by also checking for **user intent cues**: for instance, if the user performed a *modified click* (Ctrl+Click or middle-click), that explicitly indicates they **wanted** a new tab. In such cases, NavSentinel should automatically allow the new tab (avoiding false positives). Handling these cases (e.g. command-click on Mac) is important – note that other extensions had to patch this later [11].

Content Script Interception (Main vs. Isolated World): The plan to use a content script in the **isolated world** for safe observation and one in the **page's main world** for patching is appropriate. To strengthen it, ensure the script runs at `document_start`. This early injection lets NavSentinel override navigation functions before any site script can fire. For example, overriding `window.open` at `document_start` means any later calls (on click events) go through NavSentinel's wrapper. As the *No New Tabs* developer found, earlier versions only removed HTML targets but missed script-driven popups; the solution was an overhaul to intercept script calls too [5]. NavSentinel should override

`window.open` (and perhaps `Window.prototype.open`) to intercept **all** open attempts. Similarly, intercept link clicks: add a capturing `click` event listener on the document (already planned via pointerdown capture). On a capture-phase click, NavSentinel can decide to `preventDefault()` if the link looks deceptive or if a token is missing. This covers cases like `<a target="_blank">` links (which are otherwise automatic new tabs) [7]. In later stages, monkey-patching form submissions (`HTMLFormElement.prototype.submit`) or location changes (`window.location` setter or `window.open` in iframes) may be needed to handle same-tab redirects and more complex pop-unders. Designing the code to allow **plugging in these extra patches** later will create room for improvement as new vectors are discovered.

Token Correlation and Messaging: With Manifest V3's service worker background, NavSentinel should minimize round-trips for performance. The content script can do most checks synchronously (since blocking a navigation must be immediate). For example, on `window.open`, the injected wrapper can check a global token object (set by the last user click) to decide block/allow. Only if a **prompt** is needed would it message the service worker or a UI component. One approach is to have the content script itself render a small in-page prompt (e.g. a fixed overlay or a tooltip at the click location) asking "Allow this navigation?" with options for once or always. This avoids the latency of creating a full extension popup or system notification, and keeps the user experience fluid. However, a challenge is styling this prompt so it isn't hidden by the page's own elements. Using an **isolated world** injection for the prompt (e.g. an external shadow DOM or high z-index element) can help ensure the page can't easily tamper with or hide it. The prompt should include the destination URL (so the user can see if it looks suspicious) and buttons for **Allow once** or **Always allow** (which would add the site or destination to an allowlist). The service worker can manage this allowlist and send updated settings to the content script (e.g. via `chrome.runtime.sendMessage` or by the content script checking stored settings on each nav attempt). Designing a clear messaging protocol between content scripts and the service worker is key. For instance, on a nav attempt with an uncertain score, the content script pauses the navigation (does not call the real `window.open` yet), notifies the background or directly shows the prompt. If the user clicks "allow once", the content script then invokes the real navigation; if "always allow [site]", the background adds it to allowlist storage (so future attempts skip prompting) and then navigation proceeds.

Handling Edge Cases: Malicious sites often use clever tricks, so NavSentinel's architecture should anticipate them:

- **Invisible Overlays:** The plan to use `elementsFromPoint()` at click time is great for detecting transparent overlays that cover the UI. If the top element at the click point is not what the user likely intended (e.g. it's nearly transparent or off-screen sized), NavSentinel can flag a high CDS. One improvement is to momentarily set `pointer-events: none` on suspect overlay elements during hit-testing (as some research suggests [12]) to see what's truly underneath. However, be careful to do this *after* capturing the click (so the overlay still receives the click if it's legitimate).
- **Rapid Multi-popups:** If a single gesture triggers multiple `window.open` calls (e.g. one click opening 2+ tabs), NavSentinel should flag this as very high risk. The design already mentions counting multi-attempts per gesture; implement this by invalidating the gesture token after the **first** navigation attempt, so subsequent opens lack a token and can be blocked automatically. The extension *Popup Blocker (strict)*, for instance, queues multiple requests in a list for the user [6] [13]; NavSentinel can instead auto-block extras as "unwanted".
- **Bypass via iFrames:** Some scripts attempt popups from within iframes (sometimes invisible ones) that might be outside the extension's immediate control. Ensure NavSentinel's content script is injected into all frames (by using `"all_frames": true` in the manifest for content scripts on appropriate URLs). For cross-origin iframes where injection isn't allowed, consider a

fallback: Chrome's `webNavigation.onCreatedNavigationTarget` event can detect when a new tab is opened from a tab. NavSentinel's background could listen to that and, if the source tab had no valid gesture token at that moment, immediately **close the new tab** or at least mark it for blocking. (This is a last resort, as it may cause a brief flash of a new tab.) Using Manifest V3's **declarativeNetRequest (DNR)** in Stage 5 is another idea: for example, when a navigation is deemed malicious, a dynamic DNR rule could temporarily block that specific URL or domain from loading [14]. DNR could also help block known ad domains outright, though NavSentinel's privacy-first stance means no giant blocklists by default. Still, modularity here means you could let advanced users import a list of "known bad domains" to auto-block (as an optional improvement down the road).

Performance Considerations: The architecture as proposed is mindful of performance: only reacting to user gestures and navigation events, rather than scanning the DOM continuously. Each click triggers O(1) checks (hit-testing a few elements, computing style attributes) which is negligible in modern pages. A tip is to avoid any **layout thrashing**: use `getBoundingClientRect` or `elementsFromPoint` sparingly and only at needed moments (which you are – on click). Also, reuse the GestureToken approach to avoid recomputation: for example, store the computed CDS with the token, so that when a navigation uses it, you don't recalc the deception score from scratch unless needed. Memory overhead is low (just small objects per click). Using pure content script logic for decisions means the extension doesn't incur expensive messaging or background wake-ups on every click – only when a prompt is needed. This helps keep things fast and ensures NavSentinel doesn't become a bottleneck.

Security Considerations: NavSentinel's plan to not perform any network calls or content exfiltration is good for user trust. All analysis is local. One thing to watch is that by injecting a script into the page's main world (to patch functions), we must ensure not to introduce new vulnerabilities. Keep the injected script as minimal and **self-contained** as possible. For instance, it should not expose any new globals or be susceptible to the page overwriting NavSentinel's functions. Consider sealing your hooks – e.g. store the original `window.open` in a closure and override it with a wrapper that cannot be easily bypassed. (Some popup scripts try to hold a reference to the original `window.open` before extensions override it [15] [5]. To mitigate that, NavSentinel's wrapper might need to detect if the original is being called indirectly. This can get tricky; a simpler approach is to override `Window.prototype.open` as well as the `window.open` property, making it harder for scripts to grab a real handle. The test harnesses available for popup blockers [16] [17] can guide you in plugging these holes.)

Finally, isolating NavSentinel's decision logic (risk scoring) from the DOM-specific code will make the architecture cleaner. For example, have a module or set of functions that take in the gesture and nav features and return Allow/Block/Prompt. This could later be swapped out or tuned (e.g. updating thresholds or adding new heuristic signals) without touching the low-level event capture code.

Technology Stack and Implementation Recommendations

NavSentinel will be implemented as a **Chrome/Edge Manifest V3 extension** (and likely compatible with Firefox's WebExtension API with minor tweaks). The core stack is web technologies – here's what will work best:

- **Language:** Use **TypeScript** for the extension scripts (content scripts, background service worker, and options page). This adds type safety to catch errors early and makes complex logic (like scoring algorithms and message passing) easier to manage. Many successful extensions use TypeScript now for reliability.
- **Build Tools:** Since MV3 disallows certain bundling pitfalls (like synchronous `eval` or dynamic imports without enabling `unsafe-eval`), choose a bundler that's extension-friendly. Tools like

Webpack (with appropriate config) or **esbuild/Vite** can bundle your multiple scripts into few files for efficiency. You'll likely have separate entry points: one for content scripts, one for the background SW, one for the options UI. These can all live in a monorepo and be built together.

- **Frameworks:** For the **Options UI** (and any user-facing interface like a popup menu or prompt if you decide to use extension popups), a lightweight framework can help. Since the UI is relatively simple (some settings toggles, list management for allowlist, maybe a log of blocked navs), you could use **React** or **Svelte** for convenience, but plain HTML/Vanilla JS is also fine if you prefer to avoid overhead. If you do plan a richer UI (with tabs, filtering through logged decisions, etc.), a small framework might speed development. For example, **Svelte** compiles to very small, fast code, which is great for an options page that shouldn't bloat the extension. React is heavier but if you're comfortable with it, it's viable given the options page isn't performance-critical.
- **Testing:** It's fantastic that a deterministic "Gym" is planned. You might use **Playwright** (as mentioned) or **Puppeteer** to automate Chrome with the extension loaded, clicking through the test levels. Additionally, write unit tests for the scoring logic in isolation. You can use a testing framework like **Jest** or **Mocha** with a simulated DOM (JSDOM) to feed in synthetic click events and ensure the GestureToken and scoring behave as expected. This way, core logic can be validated quickly. Integration tests with real browsers will then ensure the extension works in practice.
- **Browser APIs:** Rely on the **Chrome extension APIs** where appropriate – e.g. `chrome.storage.local` for persisting settings (with perhaps `storage.sync` for allowlists if you want them synced across devices). Use `chrome.runtime.onMessage` for content script ↔ background communication, or consider the new MV3 `chrome.scripting` API if you need to dynamically inject scripts (maybe for future stage handling of form submits or so). The declarativeNetRequest API can be included in the manifest with an initial static rule set (perhaps empty or just a placeholder for future use in Stage 5). Keep in mind MV3 service workers can **suspend** when idle; ensure any long-lived logic (like awaiting user input) is handled carefully. For example, if you show a prompt in content script and wait for user click, the background might go to sleep – so if the user chooses "always allow", the content script should handle updating `chrome.storage` directly or ping the background to wake it. In practice, sending a message will wake the service worker, so that's fine. Just be sure to account for this in design (e.g. don't assume a background script can hold state in memory for too long; better to commit decisions to storage promptly).

Overall, the tech stack is fairly straightforward for a Chrome extension – the key is structuring the code for clarity: perhaps separate files/modules for **token logic**, **scoring logic**, **UI components**, and **patching hooks**. A well-organized TypeScript project will make NavSentinel easier to maintain and extend as you implement later roadmap stages.

Future Improvements and Evolution

NavSentinel's roadmap already outlines logical stages, but further ideas can strengthen it and provide **room for improvement** down the line:

- **Adaptive Risk Model:** Over time, you might refine the risk scoring. For instance, incorporate a small element of **behavior learning** – if a particular site or script frequently triggers prompts and the user always clicks "Allow", NavSentinel could learn to adjust the threshold or auto-allow that scenario. This could be done carefully to avoid security regression (perhaps after X repeated allows in a short period, treat that navigation as benign for the session). Machine learning is likely overkill (and privacy-sensitive), but simple heuristics or user feedback loops can improve UX.

- **Crowd-Sourced or Maintained Allow/Block Lists:** While the extension should work offline, an optional feature could let users contribute to a community-driven list of known deceptive domains or, conversely, known trusted domains. For example, many users of adblockers maintain filters of common "clickbait redirect" domains. NavSentinel could allow importing such a list to automatically block *those* outright. Conversely, major sites that use popups legitimately (payment gateways, OAuth login pages) could be pre-included in a default allowlist so that their new tabs don't even prompt. This must be handled cautiously and transparently, but it's a potential improvement for user convenience.
- **User Interface Enhancements:** The prompt mechanism could be enhanced with more info and control. Down the line, a **log page** in the options could show recent blocked or allowed navigations and the reasons (CDS/NRS components) for each decision. This would help power-users understand NavSentinel's decisions and fine-tune settings (e.g. "NavSentinel blocked `example.com/offer` because the click was on an invisible overlay and it was a cross-site popup"). Providing this transparency can build trust and also help during development (users might report false positives with context from the log).
- **Same-Tab Redirect Control:** Stage 4 mentions tackling same-tab redirect vectors. This is challenging because virtually any click that goes to a new page in the same tab is hard to "block" without breaking legitimate navigation. One idea is to **monitor rapid redirect chains**: if a page, right after loading, quickly redirects multiple times (especially across domains), NavSentinel could intervene (perhaps by showing a prompt "This page is redirecting you through multiple sites, continue or go back?"). Another approach is to intercept `window.location.assign` or `history.pushState` calls that occur within a short time after a gesture, if they go to a different origin than expected. Implementing this would require careful testing to not interfere with benign flows (some single sign-on flows do intermediate redirects, for example). The key is to extend the gesture token concept: a user click might be tied to a chain of navigations; NavSentinel could propagate the token through redirects for a limited time or number of hops, and if a redirect occurs without a token (or after token expiry), treat it as suspect. This is complex but doable incrementally – leaving room in the architecture to plug in such logic later is wise.
- **Collaboration with Built-in Browser Features:** In the long run, if NavSentinel proves effective, it might inspire changes to browsers' own behavior. For now, as an extension, you might explore using Chrome's internals where possible – e.g., Chrome's popup blocker icon could potentially be harnessed or replaced with NavSentinel's messaging (though Chrome doesn't give extensions direct control over that icon). However, you can use extension icons/badges of your own: for instance, flashing the browser action icon or a badge count when something is blocked, to draw user attention if they missed the prompt. This is a minor enhancement, but it improves visibility.

Finally, maintain an open test suite. The "Gym" levels 1–9 cover a great range of scenarios; you should continue adding to these as you encounter new techniques (the popup test page [16] [18] lists many creative methods – use it to verify NavSentinel handles them all). Automated end-to-end tests will ensure that new features (or Chrome changes) don't break the protection – essential for a security-oriented tool.

Conclusion and Plan Consolidation

In summary, NavSentinel's plan is solid and addresses a real pain point in web browsing. By using **user gesture tokens** and risk scoring, it strikes a balance between **strict protection** and **usable flexibility** – something current one-size-fits-all blockers lack. Grounding the plan in existing research and solutions: we know deceptive click tactics are widespread [3] and that similar browser extensions have had to

evolve to catch ever more cunning popup methods [5]. NavSentinel will build on these lessons with a cleaner architecture:

- A dual **content script** system (isolated for observation, main-world for intervention) to intercept clicks and nav attempts at the earliest point.
- A **service worker** background to handle persistence (settings, allowlists) and possibly backstop blocking (using `declarativeNetRequest` or tab management for fringe cases).
- A thoughtful **UX** that only involves the user when necessary (prompt on uncertainty) and learns from their choices.

By choosing a robust web-tech stack (TypeScript, modern bundling, comprehensive testing) and keeping the design modular, the project will be maintainable and ready to integrate new detection techniques. As development progresses, continuously revisit the threat landscape – if attackers find new ways around the defenses, NavSentinel's architecture should allow adding new countermeasures (for example, patching new APIs or adjusting scores) without a complete rewrite. This forward-looking design – leaving "room for improvement" – will make NavSentinel effective not just at launch but in the long run, as a dependable **Navigation Intent Firewall** for users' browsers.

Overall, the plan is well-grounded. By incorporating the enhancements and ideas above, we can **consolidate NavSentinel's blueprint** into a formidable extension that significantly curbs malicious navigations while preserving a smooth browsing experience [6] [9]. It's an ambitious but achievable project that could very well set a new standard for how browsers handle user-initiated navigation security.

Sources: The concepts and recommendations above were informed by Chrome extension best practices and similar tools' documentation (e.g. *No New Tabs* [5] and *Popup Blocker (strict)* [6]) as well as academic research highlighting the prevalence of clickjacking and unwanted redirects [2] [3]. These references underscore the importance of NavSentinel's features and guided the suggestions for architecture and future improvements.

---

[1] [5] [8] [11] No New Tabs - Chrome Web Store
https://chromewebstore.google.com/detail/no-new-tabs/gneobebnilffgkejpfhlgkmpkipgbcno

[2] [3] [4] [10] 2020_WWW_DirtyClicks.pdf
https://sites.cs.ucsb.edu/~chris/research/doc/www20_clicks.pdf

[6] [7] [14] GitHub - schomery/popup-blocker: A reliable popup blocker with history
https://github.com/schomery/popup-blocker

[9] [13] Popup Blocker (strict) :: WebExtension.ORG
https://webextension.org/listing/popup-blocker.html

[12] [PDF] Browser Extension Clickjacking One Click and Your Credit Card Is ...
https://media.defcon.org/DEF%20CON%2033/DEF%20CON%2033%20presentations/Marek%20T%C3%B3th%20-%20Browser%20Extension%20Clickjacking%20One%20Click%20and%20Your%20Credit%20Card%20Is%20Stolen.pdf

[15] [16] [17] [18] Test Popup Blocker :: WebBrowserTools
https://webbrowsertools.com/popup-blocker/