ChatGPT

Introduction

Apple’s Screen Time API (notably the **FamilyControls** and **DeviceActivity** frameworks introduced in iOS 15+) is designed with privacy in mind. When a parent app selects apps on a child’s device to monitor or restrict, the API does **not** expose the apps’ names, icons, or bundle IDs directly. Instead, it provides opaque *tokens* (e.g. `ApplicationToken`) that represent the chosen apps. These tokens cannot be resolved to identifiable app metadata by the parent app under normal conditions ①. In other words, the parent-side app only knows “some app” was selected – it doesn’t know which one, unless it uses Apple-provided UI components to display that info in a controlled way.

This privacy-centric design means you **cannot simply decode or “rehydrate”** an `ApplicationToken` on the parent’s device to get the app’s name or bundle identifier. Developers have tried and confirmed that properties like `Application(token:).bundleIdentifier` or `.localizedDisplayName` return `nil` in the main app context ② ③. Apple’s intention is that only the system or approved extension contexts can interpret these tokens, preventing apps from quietly harvesting a child’s app list. Below, we delve into whether one can bend these rules – for example, by leveraging a shield extension out of its normal context – and we examine known workarounds, technical barriers, and any attempts by developers to work around these restrictions.

Screen Time Tokens and the Shield Extension Context

ApplicationTokens and Privacy: An `ApplicationToken` is an opaque identifier (essentially a blob of data) corresponding to a specific app that a user selected via the Family Activity Picker. The token is deliberately opaque – the main app is *not allowed* to turn it into an app name or ID on its own ①. This is confirmed by Apple’s documentation and developers’ experience: *“Your main app cannot get the names of the apps nor their identifiers. This is a deliberate, privacy-preserving feature…”* ①. The FamilyControls framework forces you to use Apple’s provided UI for selection so that the parent never directly handles sensitive child app info. After selection, you get sets of tokens (for applications, categories, or web domains), which you might store or use to configure restrictions – but they remain opaque in the parent app.

Shield Extensions: The *shield* is the Screen Time overlay that appears when an app is blocked. Apple allows developers to customize this via a **Shield Extension** (implementing `ShieldConfigurationDataSource` and `ShieldActionDelegate`). Importantly, the shield extension runs on the **child’s device** (or generally, on the device that is being monitored/limited). It is essentially a child-side context meant to present a blocking UI and handle user actions (like “Request more time”). This extension can access some app info for display because, on a child’s device, it’s running with the Screen Time entitlement in a context where the system knows which app is being blocked. For example, the shield extension is expected to show something like “<App Name> is not available” with the app’s icon – so Apple does provide a way to get that in the extension’s UI.

However, **running a shield extension on a parent’s phone (which isn’t under any restriction)** is not straightforward. The shield extension isn’t a normal on-demand library you can call into; it’s a separate extension target that the system invokes when needed (i.e., when an app is being shielded on a managed device). If the parent's device isn't actually enforcing a restriction, there's no native event to trigger its shield extension. In theory, a parent app could include the extension in its bundle, but unless the parent device uses Screen Time restrictions on itself, the extension would never be activated by iOS. Even if one tried to manually initialize or mimic the extension, it wouldn't have the proper environment or permissions to resolve tokens to names. The tokens themselves might be meaningless on a different device – they are likely tied to the child's context (possibly encrypted or scoped per family/child). Developers have observed that tokens can become invalid if the app's signing or context changes (for example, tokens generated before an app was transferred to a new developer account stopped working afterwards) [4] [5]. This suggests the token's validity and resolvability depend on specific identifiers (like team ID or device) – using them on another device or outside the intended context yields nothing (e.g. labels failing to display the name/icon when using an out-of-context token) [5].

Shield Context on Parent Device: Could a parent app pretend to be in a "shield context" to decode tokens? Practically speaking, no. The Screen Time API requires explicit entitlements (`com.apple.developer.family-controls`) and user authorization. A parent app typically requests `.individual` (for self) or `.guardian` authorization to manage a child's settings, whereas a child-side app uses `.child` authorization. The shield extension's functionality is tied to `.child` usage – i.e., it's meant to run on the child's device or any device under Screen Time restrictions. A parent phone, logged in as the adult/guardian, isn't under those restrictions, so iOS wouldn't normally run a shield UI there.

Some developers have tried creative approaches, such as marking the parent device as `.child` in code or using the same app in "child mode" on the parent device, but this doesn't grant magical access to the child's app data. Without the proper iCloud family context and Apple's Screen Time servers recognizing the device as the child's, the parent's device can't retrieve the list of the child's installed apps via the API. In fact, one forum post describes an attempt to have a parent pick a child's apps remotely – the developer expected the FamilyActivityPicker on the parent's phone to show the child's apps (as demonstrated in Apple's WWDC video), but instead it only showed the parent's own apps [6]. The likely cause was a misconfiguration or using a simulator, but it illustrates that the API is strict about context: the parent must truly be linked to a child's account and use the authorized channels to even list the child's apps. When it works, the parent's device can initiate a FamilyActivityPicker that shows the child's apps, but still, upon selection it only receives tokens, not raw names [7].

Sandbox and Data Exfiltration Limits: Even *if* one could somehow run a shield or report extension on the parent device with knowledge of the child's tokens, getting that data back into the parent app or off the device is intentionally difficult. Apple runs these Screen Time extensions in a locked-down sandbox *specifically to prevent leaking sensitive data* [8]. As one Stack Overflow answer notes, *"the extension runs in its own sandbox and is not capable of communicating over the network or moving sensitive data out of its own address space"* [9]. The extension can read data shared by the main app (via app group containers), but it cannot just call back into the parent app with the names of the apps. There's no API to say "give me the bundleID for this token" that the main app can use at will. Apple's design assumes that if an app is showing you another user's app usage, it should do so in a tightly controlled UI that cannot secretly transmit that info elsewhere.

Known Workarounds and Attempts

Despite these restrictions, developers of third-party parental control or productivity apps have understandably wanted to present friendly app names/icons in the UI or even know which apps were selected for blocking. A few known approaches and attempted workarounds include:

- **SwiftUI Label Trick (UI Display):** Apple does allow displaying the app's name and icon *in the UI layer* by using SwiftUI components. For instance, you can initialize a `Label` with an `ApplicationToken` (or category/web domain token), and it will render the real app name and icon in your app's interface [10]. This is essentially a sanctioned workaround: the system populates the Label's view with the app's metadata, but your code still doesn't get direct string or image data. Many developers have discovered this trick. In the Apple Developer Forums, an Apple engineer (or experienced developer) confirmed *"You can pass any of the tokens to the SwiftUI* `Label` *constructor to show the app name and/or icon."* [10]. This is how apps like Opal or One Sec likely show "Instagram" or other app names in their interface – they aren't decoding the token themselves, but rather leveraging Apple's UI to do it. The parent app remains blind to the underlying identifier; it simply hosts a view that the system fills in.

- **DeviceActivityReport Extension (Usage Reports):** For displaying usage data (like "last used apps" or screen time charts) on the parent's device, Apple provides the **DeviceActivityReport** extension. The parent app can embed a `DeviceActivityReport` SwiftUI view which the system populates via the extension. This extension receives aggregated activity data, including tokens for apps or categories the child used, and it can produce a read-only SwiftUI view (for example, a bar chart of app usage with labels). The key is that this extension also runs in a sandbox – it can show the data (with app names) but cannot leak it out to the parent app's logic. As Paul Hudson explained in the Stack Overflow thread, if you want a custom report screen, *"you need to implement a Device Activity Report Extension… [which] will receive* `DeviceActivityData` *for applications/websites and present it in a view"*. But again, that extension *cannot send that data back to the main app* or off-device [8]. It's strictly for on-device display. In practice, this means a parent app **can** show a list of the child's recently used apps with names/icons – by embedding the extension's view or using Apple's provided `DeviceActivityReport` view – but it **cannot programmatically iterate through those names or transmit them elsewhere**. The data lives and dies within the extension's UI. This aligns with Apple's privacy goals: the parent can see the info (which they are allowed to know as the guardian) on their screen, but the app developer or any remote server never directly sees the child's app list.

- **Rehydrating Tokens (Attempted and Thwarted):** Some developers have attempted to sidestep these limitations by manually encoding/decoding tokens or using the shield extension in unintended ways. On Apple's forums, a developer asked how to decode an `ApplicationToken` from the raw bytes (which appeared as gibberish when printed) using `Token<Application>.init(from: Decoder)` [11] [12]. The attempt was basically to treat the token as a codable object and serialize/deserialize it, hoping to reconstruct something like an `Application` object with meaningful fields. This approach failed – the output was "didn't work" every time [13]. Another forum discussion explicitly noted that *"you aren't allowed/supposed to rehydrate the app selection tokens"* outside of Apple's UI context [14]. In that thread, a developer observed that some apps (Opal, etc.) somehow show app names and guessed they might be doing something tricky. One tongue-in-cheek suggestion was that maybe those apps take a **screenshot** of Apple's selection UI or the token's Label and use OCR to get the name, then send it to a server [14]. This gives a sense of how far-fetched one would have to get – essentially hacking around the API's intent. There is no public evidence that any reputable app actually uses OCR or similar, but the fact that it was suggested shows how no straightforward API exists for this. The likely reality is that those apps simply use the UI Label technique or maintain a mapping that the *user* helps provide (for instance, the user selects "Instagram" and the app might internally tag that token as "Instagram" for display after the fact, using the Label view to be sure).

- **Using the Shield Extension on the Child's Device:** In a proper setup, the child's device will run the shield extension when a shield is applied. That extension code can know which token it's dealing with (the system hands it the `ApplicationToken` that is being blocked in `ShieldActionDelegate.handle(action:for:applicationToken:…)`). Within the shield extension's process, one could theoretically attempt to convert that token to an `Application` object. It's unclear if `Application(token:)` yields a non-nil `localizedDisplayName` inside the extension (it might, given the extension's purpose). If it does, a determined developer might try to stash that name or bundle ID into shared storage (e.g. an app group UserDefaults) for the parent app to read. This would be an *unauthorized data export*, violating the spirit of the API. Technically, the Screen Time extensions are allowed to use app group containers (Apple even recommends using them for sharing small bits of state). For example, some apps store a token in UserDefaults from the shield extension to know which app to un-block later [15] [16]. Thus, one *could* attempt to smuggle an app's identity out by writing to the group container. **However, there are barriers**: the extension might not have any better API to get the name than the main app does (most likely it would still rely on SwiftUI to display the name, rather than give it to you raw). Also, abusing the extension this way would likely violate Apple's App Review guidelines – if discovered, it could lead to rejection or revocation of the Family Controls entitlement. Apple is quite strict, requiring developers to apply and be approved for the Screen Time entitlements, and they monitor that you use them for the intended purposes (parental controls, with privacy). Any "creative abuse" (like funneling child app data out of the sandbox) would risk that approval. In practice, we haven't seen reports of successful token rehydration via shield extension tricks. The prevailing sentiment in developer forums is that *if* some app appears to know the app names, they're either using allowed UI mechanisms or the user's own input – not a secret API.

- **Reverse Engineering Tokens:** Another angle is whether the token data itself can be reverse-engineered. The tokens are 128-byte values (as seen in debug prints [17]) that likely encodes the app's identity in some encrypted form. There's no documented algorithm to decode them, and given that they can randomly change (one developer reported Screen Time "suddenly provides new, random, unknown tokens" occasionally [18] [19]), trying to map them to apps is unreliable. In one sec's developer blog, Frederik Riedel lists as an issue that there's *"no way to open target app from an ApplicationToken"* – highlighting that without a direct mapping to a bundle ID or deep link, you can't even programmatically launch the app from the token [20] [21]. This reinforces that the token is intentionally opaque; Apple didn't provide a method to get, say, a URL scheme or identifier from it (which would have been the backdoor to identifying or opening the app).

- **Using MDM or Other Channels:** Some developers initially speculated that apps like Jomo or Opal might use MDM profiles or private APIs since they seemed to achieve things beyond the public Screen Time API [22] [23]. However, those apps have stated they do not use MDM or VPNs. Instead, they likely innovate within Apple's frameworks: for example, Opal might use the DeviceActivityMonitor trick to estimate screen time (firing periodic threshold events to count minutes of usage) [24] [25], and use the Label trick to display app names. So far, no one has found an *officially supported* way to get the actual bundle ID or name in code on the parent side.

Technical and Policy Barriers

Entitlements and Signing: The FamilyControls API requires a special entitlement that Apple grants after a review of the app's purpose. This is not enabled by default. Without it, the APIs simply won't authorize or do anything useful. The need for this entitlement means Apple knows who is using Screen Time capabilities. They can and do enforce rules around it. For example, developers often wait weeks for approval to use these APIs [26], and even after approval, there have been issues (like features working in internal testing but not for external TestFlight users if anything is misconfigured or if Apple's system glitched) [27] [28]. This gating by entitlement prevents a rogue app from sneaking into the App Store and secretly reading a device's app list via Screen Time – if you weren't explicitly granted access, you can't even compile a functioning build with those frameworks.

Sandboxing and App Review: As mentioned, the extensions run with no network access and limited I/O. Apple's App Review also reviews the functionality of these parental control apps quite closely (there's a history of Apple being stringent with Screen Time apps). If an app were somehow extracting the child's app names and sending them to a server, that would likely be flagged as a serious privacy violation unless the user is fully aware and it's fundamental to the app's purpose. Given the spirit of the API, Apple expects that the *only* place app names/icons appear outside of Apple's own Settings is within the UX of the parent control app that the parent directly sees. They want to prevent, say, a parent app surreptitiously building a database of all apps a child has installed.

Token Scope Limitations: Tokens appear to be scoped per developer and family. The forum post about transferring an app to a new account showed that tokens collected under the old team ID no longer worked under the new team's app build [4]. This implies an `ApplicationToken` might encode the developer's identifier or some salt such that it can't be reused by another app or context. Likewise, a token from Child A's context might be meaningless in Child B's context or on the parent's device unless Apple's system purposefully allows it. When a parent picks apps for a child using FamilyActivityPicker (when it works properly), the system likely fetches the child's app list behind the scenes and provides tokens that *both* the parent's app and the child's device will recognize (the parent gets an opaque token which it can send to the child's device or use in a DeviceActivity schedule). But the parent's local environment still can't decipher it — it's just a pass-through identifier. Think of it as the parent holding a key that only the child's device (or Apple's Screen Time service) can unlock to know which app it refers to.

No Official "Shield on Parent" Mode: Apple hasn't provided any method to instantiate a shield UI arbitrarily. The shield UI is tied to the act of shielding an app. A parent device can certainly use the **ManagedSettings** API on *itself* (for example, a self-control app might shield YouTube on your own phone after X minutes). But that's the parent phone acting as the "child" to itself. Even in that scenario, if you tried to feed it a token that came from a different device (your child's token), it likely wouldn't work unless the bundle identifier matches an app installed on the parent's phone and the token format is recognized. If the parent also has the same app installed, there's an off chance that applying a shield with that token on the parent's device could trigger a shield (since the token might effectively encode the bundle ID and maybe some static identifier). But if it did, what would that achieve? You'd briefly shield the parent's own copy of the app, and the extension on the parent side would show the app's name (since it's installed locally). You still haven't extracted any info that you didn't already know (the parent presumably knows they have that app). Moreover, if the parent didn't have the app installed, the token would correspond to nothing on that device – no shield would apply. In summary, using the shield extension on a non-restricted device as a "decoder ring" isn't a viable strategy.

Conclusion

In conclusion, an iOS parent-mode app **cannot force itself into a true "shield context" to reveal a child's app names or icons** in a way that breaks Apple's privacy design. The Screen Time APIs are intentionally built to prevent this kind of data from leaking across contexts. The parent app must live with opaque tokens and use approved mechanisms (like Apple's picker UI, SwiftUI Label, or DeviceActivityReport extension) to present app information in a user-facing way. These mechanisms allow the parent **to see** the app names/icons on their screen but stop short of giving the app raw access to that metadata.

All attempts to circumvent this – decoding tokens manually, running extensions out-of-context, or tricking the system – run into technical roadblocks or Apple policy roadblocks. As one developer succinctly put it, *we are "not supposed to rehydrate" those tokens* outside the sanctioned UI [14]. The system's safeguards (entitlements, sandboxing, opaque identifiers) make any unauthorized use either impossible or risky.

Developers in the Screen Time community have largely accepted these constraints. Instead of hacking the shield context, they focus on user-facing solutions: for example, using the tokens to enforce limits (which the system honors) and using the provided UI components to keep the parent informed (without violating privacy). The result is a balance: the parent can manage the child's device with Apple's help, but neither the parent app nor its cloud backend can easily compile a list of the child's apps.

In summary, **no known safe or Apple-sanctioned method exists to extract app names/icons from a child's device by mimicking a shield extension on the parent's device**. The only viable approaches are the official ones – utilizing Apple's frameworks as intended. Any "creative" workaround that truly pulls restricted metadata out would likely break soon (if it even worked at all), or break the rules, given how the Screen Time system is engineered. [1] [9]

Sources:

- Apple Developer Documentation and Forums on FamilyControls and Screen Time API
- Stack Overflow Q&A: *"Swift using Family Controls to limit apps and get name of app"* – Paulw11's answer explaining opaque tokens and the need for a DeviceActivityReport extension [29] [30]
- Apple Developer Forums: *"How to get the bundleIdentifier or app name from applicationTokens?"* – discussion of token limitations and the SwiftUI Label technique [14] [10]
- Developer blog (Frederik Riedel) on Screen Time API issues – confirms lack of direct token→app mapping and other limitations [20] [21]
- Sergio Mattei's article on Screen Time data – illustrates how locked-down the data is and how apps resort to indirect methods for usage info [31] [24]

---

[1] [8] [9] [17] [29] [30] ios - Swift using Family Controls to limit apps and get name of app - Stack Overflow
https://stackoverflow.com/questions/79286261/swift-using-family-controls-to-limit-apps-and-get-name-of-app

2 3 15 16 22 23 27 28 Family Controls | Apple Developer Forums
https://developer.apple.com/forums/tags/family-controls?page=5&sortBy=lastUpdated

4 5 6 7 Managed Settings | Apple Developer Forums
https://developer.apple.com/forums/tags/managed-settings?page=4&sortBy=newest

10 14 How to get the bundleIdentifier or... | Apple Developer Forums
https://developer.apple.com/forums/thread/782492

11 12 13 Creating ApplicationToken with Dec... | Apple Developer Forums
https://developer.apple.com/forums/thread/762441

18 19 20 21 Apple’s Screen Time API has some major issues | riedel.wtf
https://riedel.wtf/state-of-the-screen-time-api-2024/

24 25 31 A curious way to grab Screen Time data on iOS · matteing.com · matteing.com
https://matteing.com/posts/a-curious-way-to-grab-screen-time-data-on-ios

26 Family Controls | Apple Developer Forums
https://developer.apple.com/forums/tags/family-controls?page=3&sortBy=newest