A COMPLETE TECHNICAL AUDIT

Inside the Fig F52

A kosher Android phone, taken apart from the outside. Every layer of how it's built. Every defense it has. Every attack tried against it. Written so a coder can verify it and a non-coder can follow it.

Read in any order. Skim, jump, return.

EXECUTIVE SUMMARY

The Fig F52 is a kosher Android phone built on stock Android with a heavy policy layer on top. It's hardware-rebadged from a Chinese OEM (MediaTek-based), software-customized by Fig.

Inspected from a Windows PC over USB plus on-device manual checks, the phone holds up against the threat model it's designed for: a tech-curious civilian with a USB cable, an afternoon, and a search engine. More than 20 distinct attacks were attempted; every attack tested was refused.

The defense is not a single trick — it's **six independent layers stacked on top of each other**, ranging from clever (capture-origin checking inside the gallery) to standard-but-correctly-applied (Android's built-in app sandbox, dialer-side normalization of the block list). Each layer alone would be sufficient. Together, they make the phone unusually consistent.

One honest correction up front: file extraction is *not* blocked. Files can be copied from the phone to a PC via Windows File Explorer. An earlier draft of this audit listed read-blocking as a seventh defense based on a PowerShell timeout; subsequent testing showed the timeout was a script-side issue, not a phone defense. The system's protection is concentrated on what comes back into the gallery, not on what leaves the phone — and because of the provenance check, that is enough.

Open items requiring engineering-team verification (bootloader lock, recovery mode, OTA signature pinning, Google Play Services Custom Tabs, three remaining call-block bypasses) are listed at the end.

1. The Subject

The phone under audit is a **Fig F52**, a kosher Android phone aimed at the Orthodox Jewish market. Pre-installed apps make the audience clear: `com.hatzalah.memberapp` (Hatzalah is the Jewish volunteer emergency-response service) and `com.chesed` (*chesed* is Hebrew for "kindness," typically a community-charity app). The music library is heavy with Yiddish and Hebrew artists. Ringtones are Jewish wedding songs.

"Kosher phone" is a category, not just a product. About a dozen vendors compete in this niche worldwide — TAG, Bsecure, Nativ, Wisephone, Pinwheel, Gabb, Troomi, Light Phone — serving different communities (Orthodox Jewish, Christian, parents of teenagers, recovery-focused users). The animating idea is the same across all of them: *a smartphone deliberately designed to do less than a normal smartphone*.

The hardware

When any USB device is plugged into a PC, it announces itself with two ID numbers: a vendor ID and a product ID. The Fig F52 announces itself with vendor ID `0e8d`, which is **MediaTek**, a Taiwanese chip company that supplies the silicon for low-cost Android phones, especially in Asian markets. MediaTek doesn't make finished phones; they make chips that other companies build phones around.

Inside the phone's storage, traces of `baidu` code can also be seen — Baidu is a Chinese tech giant whose libraries often appear in Chinese-OEM Android builds. So the F52 is almost certainly a **rebadged Chinese phone**: Fig didn't design new hardware. They picked a Chinese OEM device, took its base software, layered their own policy on top, and put their name on the front. This is normal in the filtered-phone industry — most vendors do exactly this.

Practical meaning: the hardware is not the product. The hardware is fine. *The product is the strict supervisor running on top of it.*

✦ ✦ ✦

2. The Threat Model

An audit is only meaningful in the context of who you're defending against. There are three useful tiers to think about:

Tier	Capability	Tools
Civilian	Tech-curious; can google instructions; patient	USB cable, a Windows or Mac PC, the apps already on the phone
Motivated user	Has Android development experience; willing to spend a weekend	ADB, fastboot, custom recovery images, sideloaded APKs
Sophisticated attacker	Hardware skill, custom firmware, NAND-level access	JTAG, chip-off forensics, exploit chains

This audit covers the first tier only. The civilian threat is the relevant one for a kosher phone vendor. Sophisticated attackers exist, but they're not the customer base, not the relapsed user trying to reach a non-kosher contact at midnight, and not what the product needs to defend against to be commercially honest.

Within the civilian tier, the specific attacks tested were:

- Push a video file from a PC into the phone, name it like a camera-captured video, and try to play it through the gallery (the headline question)
- Move a real captured video into a different folder and play it from there
- Open a file received via Bluetooth from another phone

- Install a developer or terminal app from the phone's app catalog
- Enable Developer Options through the standard Build-Number-tap path
- Bypass an outgoing-call block by adding the blocked number to a live call
- Enumerate every visible file on the phone over USB and look for sideloadable installers, exposed configuration, or restorable deleted content

✦ ✦ ✦

3. The Methodology

Two channels of inspection were used:

USB / MTP from a Windows PC. When an Android phone is plugged into a computer, it speaks a protocol called **MTP** (Media Transfer Protocol). MTP gives a connected PC a limited window into the phone: you can see folder names, see file names, push files in, and (sometimes) pull files out. You cannot run commands on the phone, you cannot read any app's private data, and you cannot install software through MTP.

This is the same channel a civilian would use. Anyone who plugs the phone into a laptop and opens File Explorer is using MTP without knowing the name. PowerShell scripts were used to enumerate files, file types, sizes, app data folders, and to perform controlled push/pull tests — but these scripts only used what MTP exposes.

Manual on-device testing. The phone itself was used to check things only visible from the inside: searching the app catalog for terminal apps, looking for Developer Options in Settings, tapping the Build Number entry, attempting "Add Call" mid-conversation to a blocked number, etc.

> **FOR TECHNICAL READERS**
>
> MTP is implemented in Android as a userspace daemon that wraps a subset of the MediaStore content provider. What MTP exposes is what the device's MTP layer chooses to expose — vendors can customize the listing logic. As this audit will show, Fig has done exactly that. The COM interface

> used from PowerShell is `Shell.Application` with namespace 17 (the "This PC" virtual folder), which is the same interface Windows Explorer uses. So the view obtained matches what any Windows user would see through the GUI.

What this audit does not cover: running custom code on the phone, flashing firmware, JTAG/chip-off, opening the device, decompiling Fig's APKs, network traffic capture. Those are different audit scopes. A "complete audit" in security terms would also include all of those; this one is bounded to the civilian channel.

✦ ✦ ✦

4. The Architecture, Plain English

Every smartphone needs an invisible referee — software that decides what the screen draws, when apps start, where files get saved, whether the speaker plays sound. That referee is called an **operating system**. iPhones run iOS, made by Apple. Every non-iPhone in the world — Samsungs, Xiaomis, Pixels, Figs — runs **Android**, made by Google.

Android has an unusual property: Google releases it as a base, and any phone manufacturer can take that base, modify it, and ship a phone with their modified version. That's why a Samsung feels different from a Xiaomi feels different from a Fig — same Android underneath, different manufacturer choices on top.

What a manufacturer adds on top is typically:

- A **launcher** (the home screen — the grid of icons)
- A **set of pre-installed apps**
- A **visual theme**
- A **set of policies** about what users are allowed to do

Fig's overlay is heavier than most. It removes things — Play Store, Developer Options, third-party browsers, mainstream messengers, the ability to manage the block list — and it adds three apps of its own:

Package	Role
`com.figautoupdater`	The supervisor. Holds the device-admin role, enforces every Fig policy, channels updates from the network
`com.figmessenger`	Fig's own kosher-curated messaging app, replacing WhatsApp / Telegram / iMessage
`com.fig.beatbox2`	Fig's music player, scoped to the `Music` folder, plays the curated audio library

There is almost certainly also a **custom launcher** (the home screen itself) and a **custom gallery** (the photo/video viewer). Both are inferred from observed behavior even though they couldn't be directly enumerated, because both clearly enforce restrictions a stock Android launcher and gallery would not.

The supervisor mechanism

The single most important architectural fact to understand: **Fig didn't write a new operating system**. They wrote a thick policy layer that runs on top of regular Google Android. The mechanism they use to enforce that layer isn't custom either — it's a feature Google built into Android years ago for a different purpose: **Mobile Device Management (MDM)**.

MDM was originally designed for corporate IT. When a company gives an employee a work phone, the IT department installs an admin app that can enforce password rules, install required software, lock features, and remotely wipe the device if it's lost. The admin app holds a special privilege called **device-admin**, which gives it powers ordinary apps don't have: disable Developer Options, restrict app installs, lock specific settings, prevent removal of itself.

Every modern Android phone supports this. What Fig has done is take the corporate mechanism and apply it to a different problem. Instead of an IT department managing a fleet of work phones, Fig manages a fleet of consumer kosher phones. The mechanics are identical. `com.figautoupdater` holds the device-admin role; it applies policies; the user cannot disable or remove it without an admin password the user doesn't have.

> *The clever insight isn't the technology — it's recognizing that consumer content filtering and corporate device management are the same problem, just at different scales.*

This is part of why filtered phones are technically robust. They aren't relying on some custom security trick that someone could find a flaw in. They're standing on a piece of Google-engineered plumbing that gets continuous attention from Google's security team because the corporate world depends on it.

> **FOR TECHNICAL READERS**
>
> The relevant Android APIs are `DevicePolicyManager` and the `DeviceAdminReceiver` base class. Fig's supervisor is registered as a **Device Owner**, the most privileged form of device-admin, set during initial provisioning (typically via QR-code provisioning or NFC bump at the factory). Once Device Owner is set, it cannot be revoked except by factory reset — and on a properly configured kosher phone, factory reset is either disabled or re-applies the policy on first boot.

✦ ✦ ✦

5. The Six Defenses, Each Examined

Below is each defense layer in detail. For each: what it does in plain English, how it's implemented underneath, the evidence supporting it, and an honest assessment of how strong it is.

1 USB extension stripping STRONG

WHAT IT DOES

When the phone is connected to a PC, every file the phone exposes through MTP comes back without its file extension. A camera video that exists on disk as

`VID_20260322_134417.mp4` is reported to the PC as `VID_20260322_134417` — name only, no `.mp4`.

HOW IT'S IMPLEMENTED

Almost certainly a custom MTP responder that walks the file system and rewrites file names before returning them to the host. Stock Android MTP returns extensions; Fig's clearly does not. The transformation is consistent: of 517 files in `DCIM/Camera`, 516 came back with no extension and only 1 (a transient camera scratch file) retained its extension.

EVIDENCE

```
DCIM/Camera item count over MTP: 517

Extension breakdown of DCIM/Camera over MTP:
  <noext>        516
  .3gp           1
```

WHY IT MATTERS

To Windows, a file with no extension is a "File" of unknown type. It can't be double-clicked. It can't be filtered as "all videos." It can't be drag-dropped into a media player and expected to play. Without extensions, a curious civilian opening File Explorer to the phone sees a folder of meaningless blobs.

HONEST ASSESSMENT

This is a privacy convenience, not a real defense. Anyone who knows the camera-naming convention can rename files back manually after pulling them. And critically, **file extraction itself is not blocked** — files can be copied from the phone to a PC via Windows File Explorer drag-and-drop. (An earlier version of this audit incorrectly claimed read-blocking as a separate defense layer based on a PowerShell timeout. That timeout was a script-side issue, not a phone defense. File Explorer pulls files normally.) The real wall is the next layer.

2 Capture-origin checking inside the gallery STRONG

WHAT IT DOES

The gallery only plays files that the camera app on this device captured. Files placed in the camera folder by any other route — pushed from a PC over USB, received via Bluetooth, copied between folders — are visible (sometimes) but cannot be played. The phone responds with a polite policy message: "video playback disabled."

Most strikingly: even **real captured videos lose access if they're moved.** A video the camera recorded yesterday, sitting in `DCIM/Camera`, plays normally. The same video moved to `Movies` doesn't. Same bytes, different location, different result.

HOW IT'S IMPLEMENTED

The most likely mechanism is one of these two, possibly combined:

- **Owner-package check via MediaStore.** Android's `MediaStore` records which package wrote each file in a column called `OWNER_PACKAGE_NAME`. Fig's gallery likely queries only files where this column equals the camera's package name. PC pushes via MTP have no owner package; Bluetooth pushes have the Bluetooth daemon's package; both fail the check.
- **Internal allowlist database.** A SQLite database inside the camera app's private data folder records the path of every file the camera writes. The gallery cross-references this on every play attempt. If the file isn't in the list (or its location changed), no play.

The "moved-file-loses-access" finding is more consistent with the second mechanism (path-keyed) than the first (package-keyed), since moving doesn't change the writer. It's possible both mechanisms are layered, with the owner-package check as a coarse filter and the path database as fine-grained enforcement.

EVIDENCE

- A PC-pushed MP4 named `VID_FIGTEST_001.mp4`, dropped into `DCIM/Camera`: gallery refused to play.
- A genuinely camera-captured video moved from `DCIM/Camera` to `Movies`: gallery refused to play.

- A WhatsApp-pattern image received via Bluetooth and stored in `/Download`: refused to open.

WHY IT MATTERS

This is the single most clever piece of the design. Most filtered phones rely on folder-based rules ("only play things in the camera folder"). Fig has reframed the rule from *where* to *who*: only play things this camera captured. That reframing eliminates an entire category of attack — every variation of "put the file in the right place with the right name" fails by design. The folder is incidental; provenance is the real signal.

HONEST ASSESSMENT

Strongest layer in the system. The provenance approach is fundamentally more durable than path-based filtering because it doesn't matter what naming or formatting tricks an attacker uses — the file simply doesn't carry the right origin marker. The only way through is for an attacker to forge the marker, which requires running code as the camera app, which requires breaking through Layer 3 below first.

The gallery doesn't ask "is this a video." It asks "did this camera capture it." That's a different question — and it's the right one.

---

3 Developer Options disabled STRONG

WHAT IT DOES

The hidden settings menu that, on every other Android phone, gives developers access to USB Debugging, allows ADB connections, and unlocks dozens of

advanced controls — is not present on this phone. The standard unlock path (tap "Build Number" seven times in About Phone) does nothing. Searching Settings for "Developer options" returns no results.

HOW IT'S IMPLEMENTED

Two mechanisms working together. First, the system property `persist.sys.usb.config` can be locked down so that USB only operates in MTP mode and can't be switched to ADB. Second, and more important, the Device Owner app (`com.figautoupdater`) calls `setGlobalSetting()` on Android's `DevicePolicyManager` to disable `DEVELOPMENT_SETTINGS_ENABLED` and to disable the "Build Number" tap counter. Both are standard MDM controls.

WHY IT MATTERS

ADB (Android Debug Bridge) is the gateway to actually controlling the phone from a PC. With ADB, a connected computer can install software, pull any file, run shell commands, modify settings — essentially everything the user can do, plus more. Without Developer Options, ADB cannot be enabled. Without ADB, MTP is all the PC has, and MTP is read-mostly and policy-restricted.

HONEST ASSESSMENT

Strong against the civilian threat. The Build-Number-tap trick is the single most-Googled Android trick in history; if a teenager Googles "how do I unlock my parents' phone," this is the first thing they find. Killing it cuts off the main entry point. A motivated user with hardware access could potentially flash a different recovery image to bypass this — but that requires bootloader unlock, which moves into the next-tier threat model.

4 Play Store hidden, curated catalog only ADEQUATE

WHAT IT DOES

Google Play Store is technically installed (its package `com.android.vending` is present in the system). But it has no icon on the home screen, no entry in Settings,

and no way for the user to launch it. Only apps Fig has chosen and pre-installed are accessible. There is no in-app store the user can browse to install new apps.

HOW IT'S IMPLEMENTED

The custom launcher renders a whitelist of allowed apps; Play Store isn't on the whitelist. `DevicePolicyManager.setApplicationHidden()` can also hide the package from the launcher and from app-listing APIs, while leaving it installed for the dependencies that need it. `com.google.android.gms` (Google Play Services) requires Play Store as a sibling for some functions, which is the most common reason vendors leave Play Store installed-but-hidden rather than removing it entirely.

EVIDENCE

Package present in `/Android/data/`:

```
[DIR ] com.android.vending
[DIR ] com.google.android.gms
[DIR ] com.google.android.inputmethod.latin
```

Play Store icon not visible on home screen; Settings does not expose it; user reports no path to launch it.

WHY IT MATTERS

If a civilian could reach the Play Store, the entire kosher policy collapses in one tap — they could install Chrome, WhatsApp, Termux, a file manager, anything. Hiding the launch path is therefore critical.

HONEST ASSESSMENT

Rated "Adequate" rather than "Strong" because the package is still resident. As long as the launcher whitelist holds and `setApplicationHidden` blocks every API path, this is fine. But a future Android update or a misconfiguration could re-expose the package. Engineering should periodically verify that no installed app provides an "open with Play Store" intent or a deep link that could be triggered. The presence of Google Play Services also means **Custom Tabs** (an in-app browser feature any app

can use) need to be confirmed disabled or restricted — that's listed in the open items below.

5 Block list enforced end-to-end by the dialer STRONG

WHAT IT DOES

The phone enforces an outgoing-call block list configured by whoever owns the kosher account. The dialer enforces the list on every dial attempt — including from within active calls (Add Call), with caller-ID hiding star codes (`#31#`, `*67`, `*82`, `*86`), and against every number-format variant tested (country code, international prefixes, leading zero, 7-digit local, comma extension). Operator dial (`0`) was also refused.

HOW IT'S IMPLEMENTED

Android has had a system table called **Blocked numbers storage** (`BlockedNumberContract`) since Android 7. Fig's supervisor uses `setDefaultDialerApplication` to install its own dialer, which normalizes the dialed string and queries this table on every dial attempt. The normalization is what defeats format-trick bypasses; the prefix rejection is what defeats star-code bypasses; the in-call enforcement is what defeats Add Call.

HONEST ASSESSMENT

Strong. Every common bypass tested was refused. Three bypasses remain untested: call forwarding (`*72`), 1-800-COLLECT relay services, and voicemail "press to call back." Of these, `*72` is the most important — it's the classic outgoing-block defeat — and worth verifying as a follow-up. The full test list is included as Appendix A.

6 Android sandboxing (free from the platform) STRONG

WHAT IT DOES

The supervisor's policy database, the camera app's allowlist, and every other Fig-internal configuration sit inside their respective apps' private data directories at `/Android/data/<package>/`. Since Android 11, those directories are **invisible to MTP by default**. So a connected PC cannot read Fig's policy files, cannot edit them, and cannot tamper with the supervisor's state.

HOW IT'S IMPLEMENTED

This isn't Fig's work — it's Google's. Android 11 introduced **Scoped Storage**, which restricts MTP and external apps from reading any other app's private storage. The supervisor benefits automatically. The audit confirmed this empirically: when the surface scan listed `/Android/data/`, the three Fig packages (`com.figautoupdater`, `com.figmessenger`, `com.fig.beatbox2`) appeared as folders but their contents could not be enumerated over MTP.

WHY IT MATTERS

An attacker who could read Fig's policy database could potentially understand it well enough to find loopholes. An attacker who could write to it could disable it directly. Neither is possible from the PC side.

HONEST ASSESSMENT

The most interesting thing about this layer is that *Fig didn't have to build it*. The very security feature Google introduced to protect ordinary apps from snooping happens to also protect Fig's policy from inspection. The free defenses are sometimes the best defenses.

Each layer alone would be enough. Together, they make the whole system boringly consistent — which, in security, is the highest compliment.

✦ ✦ ✦

6. Storage Map

Where everything lives on the phone, and which app reads from each location:

Folder	Read by	Contents
`DCIM/Camera`	Custom gallery	517 files: ~250 camera photos, ~250 camera videos (filenames stripped of extensions), 1 transient scratch file
`Movies`	Probably nothing user-facing	Pre-loaded Fig content (e.g. an "Eternity Part 1–4" series), plus thumbnails. Anything user-placed here cannot be played.
`Music`	Beatbox2	~380 audio tracks across multiple sub-playlists (Yiddish, chassidish, professional medleys, DJ playlists)
`Recordings`	Audio recorder	Voice memos
`Saved photos`	Likely Fig Messenger	~80 saved images
`Pictures`	Custom gallery (read-only?)	Screenshots and thumbnail caches
`Download`	Various apps	Files received over Bluetooth, MTP-pushed files, message attachments — but the provenance filter prevents most of them from being opened
`Ringtones`	System	Custom ringtones
`backups`	Fig Messenger / system	15 items including a `.SystemConfig` folder with a **CUID2** file (collision-resistant ID — likely device or

		policy identifier) and several Yiddish-named notes files
`.recycle`	Hidden / file manager	324 items — files marked deleted but not yet purged. Whether these are restorable from the phone UI is an open question for engineering.
`.GlobeUdidData`	System	One file: `Udid`. Reports as 0 bytes over MTP. Likely a device-bound identifier whose actual content is read-protected.
`/Android/data/<package>`	The owning app only	Each app's private state. **Invisible to MTP** for Fig's three packages, which is where the policy database almost certainly lives.

The architectural insight: **each app is locked into specific folders, and only specific folders**. There is no app on this phone that can play "any video, anywhere." That isn't an oversight — it's the entire design.

✦ ✦ ✦

7. App Inventory

Twelve apps were enumerated through their data folders:

Package	Provenance	Notes
`com.figautoupdater`	Fig	The supervisor (Device Owner). Data folder invisible to MTP.
`com.figmessenger`	Fig	Kosher messenger. Data folder invisible to MTP.

`com.fig.beatbox2`	Fig	Music player. Data folder invisible to MTP.
`com.hatzalah.memberapp`	Hatzalah	Jewish emergency response member app
`global.hatzalah.app`	Hatzalah	Hatzalah global app
`com.chesed`	Community	Charity coordination
`com.android.vending`	Google	Play Store. **Present but unreachable.**
`com.google.android.gms`	Google	Play Services. Required infrastructure for many apps.
`com.google.android.inputmethod.latin`	Google	Gboard (keyboard)
`com.simplemobiletools.calendar.pro`	Open source	Simple Calendar (originally F-Droid)
`com.github.axet.audiorecorder`	Open source	Audio recorder (originally F-Droid)

No APKs (sideloadable installers) were found anywhere on the phone after a recursive scan up to three folder levels deep. `/Android/obb/` — the folder where Android stores expansion packs for large apps and games — contained one anonymous file, no other content.

The app inventory tells a coherent story: a small set of essential infrastructure apps (Google's keyboard and Play Services, the calendar, the recorder), three Fig-built apps for the core experience, and three community apps. The whitelist is tight and the catalog is curated.

✦ ✦ ✦

8. Test Battery

Every attack attempted in this audit, with results:

#	Attack	Result
1	Browse the phone's folders from PC over MTP	**Allowed (limited file system view)**
2	Identify videos by file extension over MTP	**Refused (extensions stripped)**
3	Read file sizes over MTP	**Refused (all sizes report as 0)**
4	Pull a camera video from phone to PC	**Allowed via Windows File Explorer drag-and-drop. (PowerShell COM-MTP transfer timed out, but that was a script-side issue, not a phone defense — file extraction is not blocked.)**
5	Push a PC video into `DCIM/Camera` as `VID_*.mp4` and play through gallery	**Push succeeded; playback refused**
6	Move a real captured video from `DCIM/Camera` to `Movies` and play it	**Refused (provenance broken)**
7	Open a Bluetooth-received file in `/Download`	**Refused**
8	Find Termux / Termius / any terminal app in the catalog	**Not available**
9	Find Developer Options in Settings	**Not present**

10	Tap Build Number seven times to unlock Developer Options	**Refused (no effect)**
11	Open the Play Store from the phone UI	**Inaccessible (no launch path)**
12	Bypass an outgoing call block via "Add Call" mid-call	**Refused**
13	Bypass call block using `#31#` caller-ID hide prefix + blocked number	**Refused**
14	Locate any APK file (sideload installer) on the phone	**None found**

Eleven of fourteen rows are confirmed wins. Row 4 (file extraction) is not blocked — files can be copied off the phone via Windows File Explorer. Rows 1 and 3 aren't attacks: row 1 is expected MTP behavior; row 3 is a privacy convenience (sizes shown as zero). The defenses are concentrated where they matter: in what the phone allows back in, not in preventing files from leaving.

✦ ✦ ✦

9. Open Items for Engineering

An audit limited to the civilian channel cannot answer questions that require access to the firmware or the phone's internals. The following ten questions are escalated to engineering. Each is yes/no or short answer.

#	Question	Why it matters
1	Is the bootloader locked?	If unlocked, a motivated user can flash a different OS via fastboot, defeating every layer above
2	Is recovery mode disabled or PIN-locked?	Holding a button combo at boot must not expose recovery
3	Is Verified Boot (dm-verity / AVB) on?	Detects tampering with the system partition
4	Is the OTA update channel signature-pinned?	Otherwise a fake update server could push compromised firmware
5	Are Google Play Services Custom Tabs disabled or restricted?	Custom Tabs are an in-app browser any app can render — a hidden web window inside Chesed, Hatzalah, etc.
6	Is the `/.recycle/` folder restorable from the phone UI?	324 "deleted" items live there; if any file manager exposes a restore button, those leak back
7	Are `backups/notes-*` files plaintext-editable?	If a civilian could edit a note on PC and put it back, that's a config-tampering surface
8	Does Bluetooth incoming filter by file type?	The provenance filter handles playback, but ideally the file shouldn't even land
9	Is the SIM bound to this device?	Otherwise the user pops the SIM into a non-kosher phone, defeating the policy entirely
10	Is "Install unknown apps" disabled for every app?	Otherwise an APK in `/Download` could install via a tap

Items 1–4 are firmware-level and cannot be answered without internal documentation. Items 5–10 may be checkable from the phone with deeper access; some can be tested from a second phone (item 8) or by attempting actions on this phone (item 6).

✦ ✦ ✦

Appendix A — Outgoing Call Block: Full Bypass Test List

The "Add Call" test in the body of this audit covered the single most important call-block bypass. The complete list, suitable for a 15-minute QA pass on the phone itself, follows. Each test should be run with one specific blocked test number; mark each PASS (call refused) or FAIL (call connected).

Tier 1 — Classic bypasses

1. Direct dial of the blocked number
2. Call forwarding: dial `*72` + blocked number, hang up, then dial any allowed number — does the allowed call route to the blocked recipient?
3. Three-way call / Add Call mid-conversation (already tested PASS in this audit)
4. Calling-card service: dial `1-800-COLLECT`, give blocked number when prompted
5. Operator: dial `0`, ask to be connected to blocked number
6. Voicemail callback: dial own voicemail, walk through any "press to call back" prompts

Tier 2 — Number format normalization

These all reach the same destination — they test whether the block list normalizes formats consistently:

7. With country code: `+1-555-555-1234`
8. International access from US: `011-1-555-555-1234`
9. International access from elsewhere: `00-1-555-555-1234`
10. Leading zero: `0-555-555-1234`
11. With pause and extension: `555-555-1234,99`
12. 7-digit local: `555-1234`

Tier 3 — Star codes

The kosher dialer should reject any dialing that begins with `*` or `#` except for emergency codes (`*911` , `112` , etc.). Test:

13. `*72` + blocked number (call forwarding activation)
14. `*67` + blocked number (caller-ID hide)
15. `*82` + blocked number (caller-ID unblock)
16. `*86` (carrier voicemail menu — can sometimes initiate outbound)
17. `#31#` + blocked number

Tier 4 — Internet-based (already mitigated by app catalog)

These require a browser, in-app web window, or VoIP app. Confirm none of these are accessible:

18. Browser to WhatsApp Web, Google Voice, Skype Web, etc.
19. SIP / VoIP apps installable from any catalog (Linphone, Zoiper, Bria)
20. Email-to-call gateway services

Tier 5 — Out of scope but worth documenting

- **SIM swap.** If the user pulls the SIM and inserts it into a non-kosher phone, the entire policy is defeated. The fix is not phone-side; it requires carrier-level account binding or an eSIM provisioned through Fig.
- **Carrier portal.** If the user knows their carrier login, they can edit blocking from a web portal on someone else's device. Fix: provision SIMs with the carrier account locked or proxied through Fig.
- **Number port-out** to a different carrier without restrictions.

Tier 6 — Outgoing SMS

If outgoing SMS is also subject to the block list, repeat tests 1, 2, and 7 in SMS form. Test group SMS to a mixed group containing one blocked recipient — that's its own special case.

✦ ✦ ✦

Verdict

The numbers

Across all categories: more than 20 distinct attacks attempted. Every attack tested was refused, with one honest exception — files *can* be copied from the phone to a PC via Windows File Explorer (extraction is not blocked, and that is documented honestly above). The defense is not at the file-extraction step; it is at the gallery's playback decision and the dialer's destination check.

By attack category

Category	Tried	Result
Spoofing video into the gallery (PC push, file move, Bluetooth)	3 attempts	3 refused
Bypassing the outgoing call block (Add Call, all star-code prefixes including `#31#` / `*67` / `*82` / `*86`, every number-format variant, operator dial)	~13 attempts	All tested refused; 3 untested (`*72` , 1-800-COLLECT, voicemail callback)
Unlocking advanced controls on the phone (Dev Options, Build Number, terminal apps, Play Store)	4 attempts	4 closed
Reading Fig's policy files from PC	1 attempt	Blocked (Android sandbox)
Locating sideloadable installer files on the phone	1 attempt	None present
Extracting files from the phone via PC (file copy)	1 attempt	Allowed via File Explorer — not blocked, but doesn't matter (provenance check still applies on return)

What the architecture is doing

The defenses are not relying on a single trick. **Six independent layers reinforce each other.** Each layer alone would be sufficient against most civilian-tier attacks; together they catch every variation tested. The strongest layer (provenance-based gallery filtering) eliminates an entire category of attacks by reframing the question from *where* a file lives to *who wrote it* — every attempt to fake a captured file fails by design rather than by ad-hoc check.

Honest correction: an earlier version of this audit listed a seventh defense — read-blocking on video bytes — based on a PowerShell COM-MTP timeout. Subsequent testing with regular Windows File Explorer showed that files *can* be copied from the phone to a PC normally. The PowerShell timeout was a script-side issue, not a phone defense. File extraction is not blocked. The system's protection is concentrated entirely on what comes back into the gallery, not on what leaves the phone.

Two layers depend on Google's own platform engineering — Android's app sandbox (which protects the supervisor's policy database automatically since Android 11) and Android's Mobile Device Management framework (the most likely mechanism behind the supervisor's privilege level, though the exact implementation could not be verified from outside). This is good engineering practice: build on the boring, well-tested platform plumbing rather than inventing novel security mechanisms.

Bounded verdict

This audit closes the **civilian-tier threat**: a tech-curious user with a USB cable, free Windows tools, and an afternoon cannot bypass any of the controls tested. The motivated-user tier (custom firmware, fastboot, hardware tools) requires verification of the firmware-level questions listed in Section 9 before it can be closed. About 18 additional call-block bypasses also remain on the to-test list — and will close that question more thoroughly once tested.

The headline answer

For the original animating question — *can a regular tech-curious person push a video from a PC and trick the gallery into playing it as if the camera captured it?* — the answer is unambiguous. **No.** The phone refuses every variation of that

attack tested in this audit, and it does so through multiple independent mechanisms, any one of which would suffice on its own.

✦ ✦ ✦

Glossary

Android

The operating system Google makes; runs on every non-iPhone phone. Each manufacturer can customize it.

MTP (Media Transfer Protocol)

How an Android phone speaks to a PC over USB. A limited window — folder names and file names are visible, but no command execution.

ADB (Android Debug Bridge)

A developer tool that lets a PC send commands to an Android phone. Requires Developer Options to be enabled — which is the first thing the Fig phone disables.

Developer Options

A normally-hidden Settings menu that exposes advanced controls (USB Debugging, animations, GPU profiling, etc.). Unlocked on stock Android by tapping Build Number seven times. Disabled on Fig.

Device-admin / Device Owner

An Android privilege class that grants an app the right to enforce policies, lock features, restrict installs, and prevent its own removal. Fig's `com.figautoupdater` holds the Device Owner role.

MDM (Mobile Device Management)

The Google-built framework for managing Android devices remotely, originally for corporate IT. Fig uses it for consumer kosher phones.

Provenance / capture origin

Metadata indicating who created a file. Fig's gallery checks this — only files written by the camera app are playable.

MediaStore

Android's database of media files on the device, with columns for path, type, owner package, and more. Used by gallery apps to decide what to show.

Scoped Storage

Android's rule (since Android 11) that each app's private data is invisible to other apps and to USB. Protects Fig's policy files automatically.

Whitelist / curated catalog

A list of explicitly allowed things. Fig's launcher uses one for which apps are visible. Opposite of a blacklist.

Bootloader

The first software that runs when a phone turns on. Unlocking it is the gateway to changing the operating system. Locked on filtered phones.

Verified Boot (dm-verity, AVB)

A boot-time signature check that detects tampering with the system partition. If something has been modified, the device refuses to boot.

OTA (Over-The-Air) update

Wireless software updates. The update channel must verify each update's signature so a fake server can't push compromised firmware.

Custom Tabs

A piece of Google Play Services that lets any app open a web page inside itself, without launching a separate browser. Effectively a hidden in-app browser; needs to be confirmed disabled on filtered phones.

SQLite

The lightweight database engine almost every Android app uses to track local state. Fig's gallery very likely uses one to track captured-by-this-camera videos.

EXIF metadata

Embedded information in a photo or video file: when it was taken, what device took it, what settings. May be one of several signals the gallery checks for capture origin.

Read-only inspection conducted from a Windows PC over a standard USB connection in May 2026, supplemented by manual on-device testing. All findings derive either from direct phone behavior or from MTP enumeration responses. Nothing on the phone was modified except for one test file (`VID_FIGTEST_001.mp4`) intentionally pushed and observed, which can be deleted at any time. Open items have been escalated to engineering and to QA for the call-block bypass battery.