ChatGPT

Reverse Engineering Techniques for iOS, Web, and Open-Source Software

Reverse engineering is the process of analyzing a software system to identify its components and how they interact, often to replicate functionality, find vulnerabilities, or modify behavior. This report provides a comprehensive guide to reverse engineering across three domains: **(1)** macOS/iOS applications (written in Swift or Objective-C), **(2)** web applications (such as React single-page apps), and **(3)** open-source software projects. Each section covers essential tools, practical methods (static and dynamic analysis), and considerations including relevant legal guidelines in the UK/EU.

Reverse Engineering macOS and iOS Apps (Swift/Objective-C)

Reverse engineering Apple platform apps involves overcoming compiled binaries and Apple's security measures. iOS apps (packaged as `.ipa` files) and macOS apps are delivered as Mach-O binaries, which are not human-readable by default [1]. Thus, the first step is often to **disassemble or decompile the binary** into a more understandable form. Below we outline the process and tools for static and dynamic analysis, the special case of iOS binary decryption, runtime method hooking, and differences between jailbroken vs non-jailbroken environments.

Key Tools for iOS/macOS Analysis: A variety of industry-standard tools are available to inspect and dissect Apple binaries:

- **Disassemblers/Decompilers:** *Ghidra* (a free open-source reverse engineering suite by NSA) and *IDA Pro* (a commercial interactive disassembler) are widely used for static analysis. Both can load Mach-O binaries and produce assembly code or pseudo-code, helping researchers understand app logic [2] [3]. *Hopper* is another popular macOS tool with a GUI that specializes in ARM and Objective-C decompilation, offering a cheaper alternative to IDA [4]. These tools let you navigate functions, examine control flow graphs, and even script analysis tasks.

- **Native Mach-O Tools:** Apple's developer toolchain includes command-line utilities like `otool` (to dump Mach-O headers, segments, libraries, etc.) and `nm` (to list symbol tables). These help reveal what architectures an app supports, which frameworks it links, and what symbols (class and method names, if not stripped) are present [5] [6]. Additionally, `codesign` can show code signature details such as entitlements and cryptographic hashes [7] [8], and `strings` can extract human-readable strings from the binary [9] – often useful for finding clues like error messages or URLs.

- **Objective-C Introspection:** *Class-dump* is a specialized tool that extracts Objective-C class information from a Mach-O binary [10] [11]. It leverages the metadata present in Objective-C binaries to recreate interface headers (class names, methods, and properties) in plain text [12]. This is extremely useful for iOS/macOS apps, since Objective-C retains a lot of runtime information. Using class-dump on a decrypted app binary yields header files that reveal the app's class structure and

method signatures, which can guide deeper analysis. (Note: Class-dump works less well on pure Swift code, since Swift's metadata is more opaque and name-mangled; Swift reverse engineering may require demangling symbols and understanding Swift runtime structures, more akin to C++ reverse engineering [13] [14].)

- **Dynamic Debuggers/Instrumentors:** *LLDB* is the debugger that comes with Xcode and can be used to step through code, set breakpoints, and inspect memory of apps running on macOS or on an iOS device/simulator [8]. On macOS, LLDB can attach to most processes (unless restricted by System Integrity Protection or code signing), allowing dynamic analysis of App Store or third-party apps. On iOS, however, App Store apps are by default protected – a non-jailbroken device will not allow attaching a debugger to an arbitrary app (the app's `get-task-allow` entitlement is false). This means that for dynamic analysis of iOS apps, one typically needs a **jailbroken device** or specialized environment to bypass those restrictions [15]. In a jailbroken setting, you can run `debugserver` on the device and use LLDB to attach to the app process, stepping through functions even without source [16] [17]. Many iOS reverse engineers install *debugserver* or LLDB directly on the device via Cydia, enabling on-device debugging over an SSH connection [18] [19].

- **Dynamic Instrumentation & Hooking:** *Frida* is a powerful dynamic instrumentation toolkit for multiple platforms that is extremely popular for mobile app reverse engineering. Frida allows injection of a JavaScript-based agent into a running process, so you can hook functions, trace method calls, and even modify return values at runtime [20]. Unlike static disassemblers, which show what the code *could* do, Frida lets you observe and change what the code *is doing* as it runs. For instance, you can use `frida-trace` to automatically trace calls to functions by name pattern – e.g., hooking any function with "jailbreak" in its name to see if an app is checking for a jailbreak environment [21]. Frida can also be used to call Objective-C runtime functions or Android Java methods directly. On iOS, Frida typically requires a jailbreak (to inject into another process), although Corellium's virtual iOS devices or certain developer-only techniques can allow it on non-jailbroken devices [22]. Other hooking tools in the iOS jailbroken ecosystem include Cydia Substrate or Substitute (which let you write tweaks that inject code into apps) and Cycript (an older tool to live-explore an app's Objective-C objects). These are beyond the scope here, but all rely on altering the app's behavior at runtime.

Static vs Dynamic Analysis: A productive workflow often combines static **and** dynamic techniques. Static analysis (disassembly, class-dumps, etc.) gives a map of the code's structure. You can identify key functions (entry points, network handlers, crypto functions) and study their implementation in assembly or decompiled pseudo-code. Dynamic analysis lets you verify these findings and dig into runtime-only behavior (like decoding data in memory or bypassing certain checks). For example, you might spot a function `-[AppDelegate isJailbroken]` via class-dump or strings. You could then set a breakpoint on it with LLDB or hook it with Frida to see when it's called, and even override its return value (making it always return false to bypass a jailbreak check). This mix of static insight and dynamic testing is the crux of modern reverse engineering.

Decrypting iOS App Binaries: A unique challenge on iOS is that App Store binaries are encrypted (to protect against piracy and tampering). The encryption is applied to the executable segment of the app when it's distributed, and is only decrypted in memory on the device at runtime [23]. This means if you simply unzip an `.ipa` from the App Store, the main binary still appears encrypted (you'll see an

unreadable Mach-O). To perform static analysis, you first need to obtain a **decrypted binary**. There are a few methods to do this:

- Using a jailbroken device with tools like *Frida's dump script* or *dumpdecrypted.dylib*. The process typically involves installing Frida on the device via Cydia, then using a Python script (such as the popular open-source **frida-ios-dump** [24] [25]) which automates the task. You launch the target app on the device, run the dump script from your computer (it uses `ssh` and Frida under the hood), and the script will inject into the running app and dump the decrypted bytes of the binary to a file. The result is a decrypted `.ipa` or binary that you can then load into Hopper, Ghidra, etc. [26]. Older tools like *Clutch* or *bagbak* achieve similar outcomes. Once decrypted, class-dump and static disassemblers can work properly on the binary [27].

- On a non-jailbroken device, obtaining a decrypted binary is harder. One approach is to use an iOS simulator build (if available) since simulator binaries aren't encrypted (but those exist only for apps you have source to, or some open-source iOS apps). Another approach is using a developer enterprise certificate to re-sign the app with the `get-task-allow` entitlement and then use LLDB to dump memory, but this is non-trivial. In practice, security researchers use jailbroken devices or services like Corellium's virtual iOS devices to get decrypted binaries. **Important:** Decrypting an app you obtained from the App Store could be viewed as circumventing a protection mechanism (raising legal considerations), though in the EU context there are allowances when done for interoperability or security research (discussed later).

Method Swizzling and Runtime Patching: Objective-C's runtime is highly dynamic – one of its powerful (sometimes dangerous) features is **method swizzling**. This is a technique that lets you exchange the implementations of two methods at runtime (essentially redirecting calls from one selector to a different function). Reverse engineers use swizzling (or more generally, hooking) to alter app behavior without modifying the binary on disk. For instance, you can swizzle a method like `- [SecureClass verifyLicense:]` to point to a dummy implementation that always returns true, effectively bypassing a license check. Swizzling leverages the loose coupling between an Objective-C *selector* (the method name) and the *IMP* (the function pointer that implements the method) [28]. The diagram below illustrates this concept: before swizzling, selectors A and B point to their original implementations; after swizzling, they have been swapped or redirected, so that calling A actually invokes the code for B (and vice versa) [28]. This can be done using the Objective-C runtime functions at runtime.

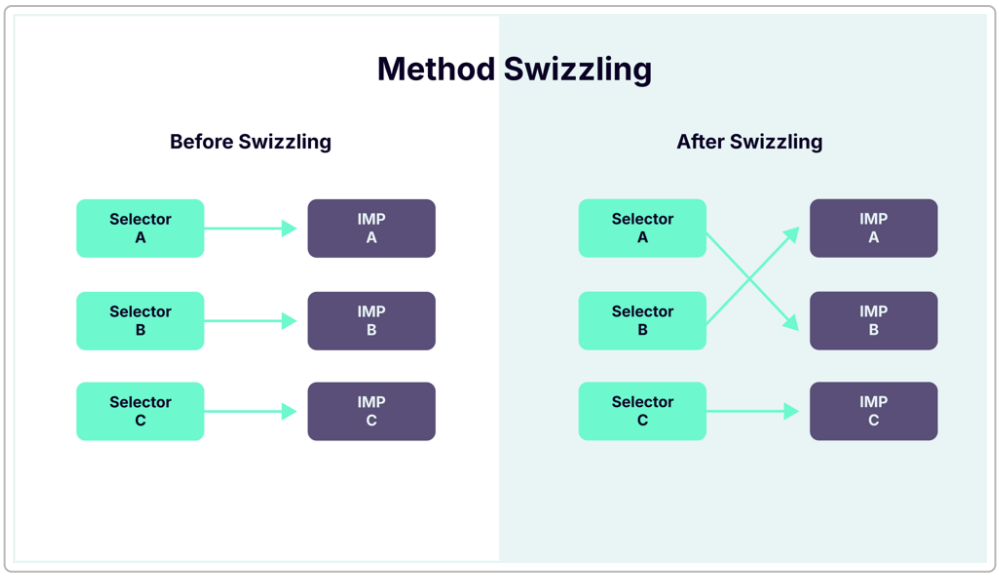


Illustration of Objective-C Method Swizzling: each method selector (name) is associated with an implementation (IMP) pointer. By swapping these pointers, a program can redirect calls from one method to another. This technique is commonly used by reverse engineers to hook or bypass functionality in iOS apps.

In practice, if you have a jailbroken iOS device, you might write a MobileSubstrate tweak or use Frida's Objective-C API to perform swizzling. For example, with Frida one can do: `ObjC.schedule(ObjC.mainQueue, () => { let cls = ObjC.classes.MyClass; let orig = cls["+ originalMethod:"]; let newImp = new NativeCallback(function () { /* new code */ }, 'void', ['object', 'pointer']); ObjC.replaceMethod(orig, newImp); });`. This replaces the implementation at runtime. Apple does not allow method swizzling in App Store apps for good reason – but for reverse engineering, it's extremely useful. (Note that Swift methods cannot be swizzled in the same way unless they are exposed via `@objc` or dynamic; Swift's direct method calls lack the ObjC message dispatch indirection.)

Jailbroken vs Non-Jailbroken Environments: As noted, many dynamic analysis techniques require a jailbroken device. On a stock (non-jailbroken) iPhone, the system sandbox and code signing will thwart most attempts to inspect or modify another app's process. You **cannot** attach LLDB to an App Store app or inject Frida into it, because the kernel will prevent it without proper entitlements [15] [29]. Jailbreaking patches the kernel to remove these restrictions (bypassing code signature checks, etc.), effectively opening the device for full introspection [29]. The downside is a jailbroken phone is less secure and some apps refuse to run on jailbroken devices (though you can often bypass jailbreak detection using the above methods!). If you cannot jailbreak (for example, if no jailbreak is available for the latest iOS), alternatives include: using an older device/iOS version that can be jailbroken, or using a platform like Corellium (which provides a virtual iOS device with jailbreak-style access). For macOS apps, the playing field is more open – macOS doesn't forbid debugging processes you have access to, and there are no encrypted binaries from the Mac App Store. Thus, reverse engineering a Mac app is often simpler: you can directly run Hopper or Ghidra on the app's binary, run the app in a debugger, etc., as long as System Integrity Protection (SIP) is disabled if you need to attach to certain protected processes. Always be mindful that modifying and redistributing Apple's code or apps can violate EULAs and laws, but analyzing them for interoperability or security is generally allowed (next section).

Relevant Legal Considerations (UK/EU): Apple's EULA disallows reverse engineering of their software, and the iOS App Store uses technical protection (encryption) to deter it. However, UK and EU law provide some exceptions for reverse engineering, especially for the purposes of interoperability and security analysis. The EU Software Directive (2009/24/EC) explicitly permits a lawful acquirer of software to *"observe, study or test the functioning of the program in order to determine the ideas and principles which underlie any element of the program"* during use [30] [31]. It also allows decompilation **if indispensable to achieve interoperability** with other software, under certain conditions [32] [33]. These rights **cannot** be overridden by any license agreement [34] [35] – meaning, in the EU a software vendor cannot enforce a EULA clause that absolutely forbids reverse engineering for legitimate purposes. In the UK, which inherited these rules from EU law, similar provisions exist in the Copyright Designs and Patents Act. That said, these laws don't give carte blanche for piracy or malicious use: the information obtained must not be used to create a competing product or violate copyright. Also, **anti-circumvention laws** (akin to the DMCA in the US) could potentially apply if one breaks encryption without falling under an exception. In practice, analyzing an app that you've legally obtained for security research (e.g. finding vulnerabilities or ensuring interoperability with your system) is likely defensible under EU law [36]. But distributing the app's decrypted code or a patched version would **not** be legal unless permitted by the app's license. Always proceed within ethical and legal boundaries: use the insights for education, security improvements, or interoperability, not for stealing intellectual property. If in doubt, seek legal advice, especially when dealing with proprietary software protections.

Reverse Engineering Web Applications (React / Client-Side Apps)

Web applications, particularly those built as rich **single-page apps (SPAs)** with frameworks like React, Angular, or Vue, present a different set of challenges. The code (JavaScript/TypeScript, HTML, CSS) is delivered to users via the browser, which means one has access to the client-side code *in some form*. However, that code is usually **minified or bundled** (e.g., via Webpack, Parcel, etc.), making it hard to read. Reverse engineering a web app typically focuses on two aspects: **(a)** understanding and possibly extracting the client-side code (JavaScript/React components, logic flow), and **(b)** analyzing the **network interactions** between the client and server (REST API calls, GraphQL, WebSockets, etc.), often to document or replicate the app's backend API. We also touch on tools for deobfuscation and legal considerations for web content.

Inspecting and Recovering Client-Side Code: Modern browsers come with excellent developer tools that are the first line of attack in reverse engineering a web app. By pressing F12 (or right-click "Inspect"), you open the **browser DevTools** which allow inspection of the DOM, scripts, network calls, and more.

- **DOM Inspection:** In the Elements panel, you can see the live HTML structure after the React app has rendered [37]. While React components don't directly show up as such in the DOM, you might find markers like `data-reactroot` or specific IDs and classes that hint at components [38]. This can give clues about the component hierarchy and dynamic parts of the page.

- **Viewing Source Files:** In the Sources panel, you'll find the JavaScript files that were loaded. Typically, a React app built with Webpack will have a few large bundle files (e.g. `main.js`, `vendor.js`, or hashed filenames). You can open these and use the built-in **"Pretty-Print"** feature (the `{}` icon) to reformat minified code into indented, readable code [39]. The variable and function names will still be obfuscated (e.g. short names like `a`, `b`, `c`), but at least the structure (loops, calls, etc.) becomes apparent. From there, you can search within these files (Ctrl+F in the DevTools source viewer) for keywords. For example, searching for `"password"` or `"login"` might lead you to the

part of code handling authentication. Searching for React-specific keywords like `React.createElement` or component names (if you know any) can also be fruitful [39].

- **Source Maps:** Many web apps include **source maps** (often inadvertently on production, or intentionally on staging sites). A source map is a `.map` file that maps the minified bundle code back to the original source files (with original variable names and all). If the site is not actively hiding them, you might see files with `.js.map` in the Network panel or listed under Sources. DevTools will often detect and use source maps automatically to show you original source code. If not, you can manually locate and load them [40] [41]. With source maps enabled, the Sources panel will show the actual React project files (e.g. `.jsx` or `.tsx` files, organized by folder) as they were during development [40]. This is the jackpot for reverse engineering: you essentially have the original code with comments (unless they stripped them) and meaningful names. Always check if `/*.map` files are accessible, as it can save enormous time. If they are, using a tool like *source-map-explorer* or *source-map-viewer* can help visualize the bundle and its source files.

- **Deobfuscation Tools:** If source maps are not available, you deal with obfuscated code. Besides manual inspection, there are tools to assist:

- **Beautifiers:** Online tools or IDE features (like VS Code's format) can prettify code (which DevTools already does). This doesn't rename variables but improves readability.
- **Name guessers:** Projects like JSNice attempt to statically analyze minified code and suggest more readable variable names using heuristics and community data. Results vary, but it can rename `a` to `isLoggedIn` in some cases.
- **AST Exploration:** Advanced users can load the minified code into an AST explorer (like using the Babel parser) to search for specific structures (e.g., find all string literals, or identify function definitions more easily than by reading raw code).

- **Browser Debugging:** You can also insert breakpoints or `debugger` statements in the code via DevTools to pause execution at interesting points. For example, pause in the function after a button click to inspect variable values (the console's scoped variables or by adding watches). This live examination can give meaning to those obfuscated variables (e.g., you see that variable `s` contains `"Invalid password"` message, revealing its role).

- **Browser Extensions:** Use the **React Developer Tools** extension (available for Chrome/Firefox). This adds a "⚛ React" panel in DevTools which shows the component tree of a React app, along with props and state for each component [42]. This is incredibly useful: you can traverse the hierarchy, see component names (which might be minified in production, but often React keeps component displayNames), and current state values. It's a great way to understand how the app is structured and what data flows through it. Similarly, there are Vue and Angular dev tools for their respective frameworks.

Analyzing Network Traffic and APIs: The behavior of a web app is deeply tied to its server communications – for example, a React app might call a REST API to fetch data. Reverse engineering often

requires mapping out these interactions (also termed **API fingerprinting** – i.e., identifying each API endpoint the app uses, what data it sends, and what it expects back).

- **Network Panel:** The DevTools Network tab will list all network requests the application makes (XHR/fetch calls, WebSocket connections, etc.). By reproducing certain user actions, you can observe the requests fired. You can filter by type (e.g., XHR) or search within the traffic. For each request, you can inspect the URL, method, headers, request payload, and response data. This is the primary way to discover REST endpoints or GraphQL queries. For instance, you might see a POST to `/api/login` with a JSON payload `{"user":"alice","pass":"..."}` and a JSON response. By logging in and performing various actions, you enumerate the app's "hidden" API. If the app uses GraphQL, you'll see typically a single endpoint (e.g. /graphql`) with query/mutation names in the payload; GraphQL introspection might be enabled, or you can glean the schema from the queries sent.

- **WebSocket Traffic:** If the app uses WebSockets (for real-time features), those will appear as a persistent connection in the Network tab (listed as "WS"). Clicking on it allows you to see messages being sent and received in real-time. DevTools will show frames, and you can often view text payloads (JSON, etc.) of each message. This can reveal undocumented protocols or message formats that the app uses for live updates. For example, a trading app might send frames like `{"op":"subscribe","channel":"ticker"}` and receive live pricing data. Tools like Wireshark can also capture WebSocket traffic (which is just HTTP upgrade + frames) if needed, but usually the browser tool suffices if not encrypted beyond TLS.

- **Intercepting Proxies:** For more advanced API analysis, or if you want to replay/modify requests, consider using a proxy tool:

- **mitmproxy** (command-line and web interface) or **Burp Suite** (popular in security testing) can act as a Man-in-The-Middle proxy. By configuring your system or browser to route traffic through the proxy (and installing its root certificate to intercept HTTPS), you can capture all requests outside the browser as well (e.g., if the app uses a separate WebSocket client or other mechanism). These tools let you see and record traffic, and even repeat or fuzz requests. Burp in particular has a powerful interface to send requests to its repeater or scanner. Mitmproxy has scripting capabilities (Python) to automatically modify or log certain patterns. There are even tools that convert captured traffic to API documentation; for example, **MitmProxy2Swagger** can analyze HTTP flows and produce an OpenAPI (Swagger) spec for the REST endpoints [43] [44].

- **Postman/REST clients:** Once you identify an API endpoint and its required parameters/auth, you can use Postman or similar REST clients to mimic the app's calls. This is part of API reverse engineering – essentially creating a collection of calls that replicate what the app does. Postman can import HAR files (HTTP Archives) exported from the browser and generate sample requests.

- **Handling Obfuscation or Protection:** Some web apps implement defenses: code obfuscation, disabling right-click/DevTools, or API encryption and request signing. For instance, some apps include anti-debugging scripts that detect DevTools and change behavior, or they require a calculated hash (API key or HMAC) with each request to prevent easy calling. Overcoming these can be difficult:

- If a site tries to block DevTools, you can often bypass it (e.g., by undocking the DevTools or using a headless browser to capture network calls).
- If requests are signed with a secret (e.g. a private key within the app), you may need to extract that key from the code. Searching the minified code for suspicious strings or long base64 blobs might reveal keys. Alternatively, hook the JavaScript at runtime: for example, override `window.fetch` or a specific crypto function to log the parameters it's called with.
- In extreme cases, one might resort to instrumenting the browser or using a phantom DOM to run the app and intercept data (for example, using Puppeteer to control a headless Chrome and intercept network events, or using browser DevTools protocol via `chrome://inspect`).

Practical Tip – Using Source Maps and DevTools: Many developers accidentally leave source maps accessible in production. Always check the site for `.js.map` references. As an example, a Medium article demonstrated retrieving a React app's source code via source maps and reconstructing the original project structure [45] [46]. If you find a `main.js.map`, you can load it in Chrome (Chrome will often prompt or auto-load if the map is linked in the JS file) and suddenly your Sources panel shows many original files. This makes reverse engineering trivial, as you can navigate the code as if you're the developer. If source maps are not present, you can still systematically approach understanding: identify initialization code (look for something like `ReactDOM.render(<App />, ...)` or in Angular find `bootstrapModule` call). That often points to the root component. Then, trace through the code, following import references (with Webpack, module names might be numbers, but often there's an array of modules in the bundle and you can find module by content search). A helpful approach is to find UI text in the app – a label or button text you see on screen – and search for it in the JS files (unless all text is external or hashed). A unique phrase in the UI can lead you right to the code that renders it.

Legal and Ethical Considerations: The code that runs in your browser is generally considered *published content* in terms of copyright – it's sent to every user's browser in a form that can be copied. There is usually an implied license for you to download and execute it as part of using the site, but not necessarily to **re-use** or redistribute it. In the UK/EU, viewing and studying the code that your browser received is likely covered by the idea that you are **observing the program in operation**, which is lawful [47] [31]. However, extracting proprietary algorithms or copying substantial portions of the code for your own project could violate copyright. Web code often lacks explicit license; assume it's proprietary unless stated otherwise. **Reusing** any client code or assets without permission is risky. On the other hand, learning from it, or using it to interact with the public API, is usually acceptable. Also be mindful of terms of service of the web app – some explicitly forbid reverse engineering or automated data extraction. While such terms may or may not be enforceable under law, violating them could at least get your access revoked or lead to other legal claims (like under anti-hacking laws if you go beyond authorized access). **API usage:** If you reverse engineer an unpublished API (say, a private REST endpoint), using it in your own app or script might breach terms of service. Yet, EU courts have held that functional aspects like APIs are not protected by copyright, and creating interoperable software is lawful [48]. So mimicking an API isn't copyright infringement per se – but beware of other issues like rate limiting, authentication, or data privacy laws if you're pulling user data. Finally, if you discover security weaknesses during your analysis, act responsibly: consider disclosure to the service owner. Reverse engineering web apps for **security research or compatibility** tends to be viewed favorably under law, as long as you don't exploit it maliciously. Still, discretion is key; do not violate privacy or publish sensitive information you might discover.

Analyzing and Patching Open-Source Software Codebases

Open-source software (OSS) provides a different scenario: you have access to the source code by definition. Reverse engineering here is less about disassembling and more about **comprehending a large codebase** that you didn't originally write. Common goals include understanding the architecture, finding where to implement a fix or feature ("patching"), analyzing dependencies, and possibly auditing for bugs. In this section, we cover best practices for exploring large open-source projects, tools for code navigation and visualization (like dependency graphs and call graphs), and approaches to patching and contributing, with notes on license compliance.

Getting Familiar with a Large Codebase: Diving into a complex project can be daunting, but a systematic approach helps:

- **Read the Docs & Setup Guide:** Start with the project's README, wiki, or developer guide. Many mature projects have high-level architecture docs or "CONTRIBUTING.md" that explain how the pieces fit. Take note of the overall modular structure (e.g., in a web server project, there might be "core", "network", "database" modules, etc.).

- **Build and Run Tests:** Try to build the project from source and run its test suite (if available). This serves two purposes: (1) ensure you can compile/run it in a dev environment, and (2) tests are excellent documentation. Reading test code can reveal how the software is *intended* to behave [49] [50]. For example, a test named `test_parse_config_file()` shows you how the config parser is supposed to work, without wading through the entire program logic.

- **Use the Software First:** Much like with open-source libraries, if possible, **use the tool/app** as an end-user to understand its functionality before reading code. If it's a library, write a "hello world" using it. If it's a GUI application, click through features. This contextual knowledge lets you map runtime behavior to code. As one guide notes, *"the best way to get familiar with any open-source project is to use it"* [51]. While using it, note what parts are confusing or likely complex – those might correspond to interesting code areas.

- **Find the Entry Point:** Every program has an entry (e.g., a `main()` function, or for libraries, a set of public API functions). Locate this in the code to get an initial anchor. For a program, from `main()` you can see how it initializes the system (parsing args, setting up subsystems). This gives a high-level call flow. In an object-oriented project, there might be an `Application.start()` or similar.

- **Trace a Typical Flow:** Identify a couple of key use-cases ("functional threads"). For example, in a web server, a key flow is "accept connection -> receive request -> process -> send response". Locate code for each stage. You might diagram it on paper. Another example: for a CRUD web app, trace the "user login" flow – from form submission, to controller, to database query, etc. This helps you understand how components interact, instead of randomly reading classes. Focusing on functionality (what does the software *do*) rather than just technical components helps build a mental model [52].

- **Examine Project Structure:** Look at the repository layout (folders and files). Often, it follows logical segregation (for instance, `/src` with subfolders by feature, a `/tests` directory, maybe `/`

`docs`, `/scripts`, etc.). Many projects use standard conventions (e.g., a Java project might use `com/example/...` packages; a Node.js project has a `lib` or `src` and maybe a `package.json`). Understanding the build system (Makefiles, Maven pom.xml, etc.) also gives insight into how components depend on each other and what external libraries are used.

- **Use Version Control History:** Sometimes the **initial commits** or history are illuminating. The first commit might have a simple version of the core functionality without all the later complexity [53]. Reading commit messages (e.g., using `git log`) can tell you why certain changes were made. Moreover, you can use git to find "hot spots" in the code: files that change often. These are usually critical parts or ones that had bugs. The command below shows the top 10 most-edited files in a git repository, which often corresponds to key functionality or tricky components (the 80/20 rule – 20% of files might contain 80% of the changes/logic):

```
# Identify top 10 most frequently edited files in the repo:
git log --pretty=format: --name-only | sort | uniq -c | sort -rg | head -10
```

(The one-liner works by extracting all filenames from commit history, counting occurrences, and sorting by count [54].)

If you run that and see, for example, that `database.cpp` and `auth.cpp` are at the top with hundreds of changes, you know those are central. Focus your attention there when trying to understand or patch things.

Tools for Code Navigation and Visualization: Unlike closed binaries, with source code you have the advantage of using code-aware tools:

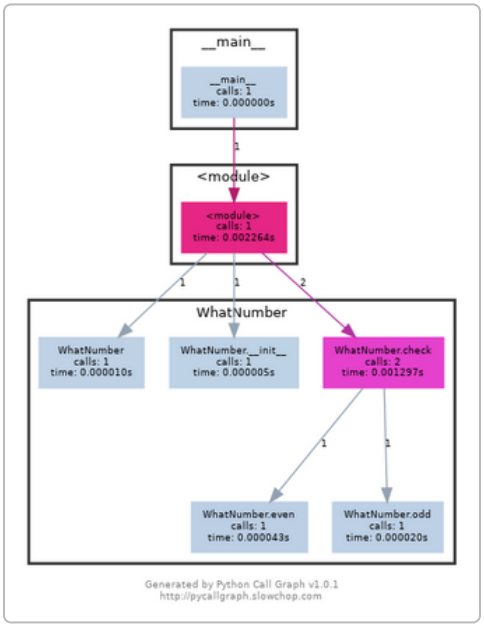
- **Text Search and Ctags:** Never underestimate the power of `grep` or your IDE's search. Searching for relevant keywords (e.g., function names, error messages, config keys) will quickly locate definitions and usages. If the project is on GitHub, you can use GitHub's inline search by pressing `/` on the repository page – it's very fast and supports regex and filtering [55]. Locally, tools like **ctags** (or the modern Universal Ctags) can index the code symbols so your editor (vim, VSCode, etc.) can jump to definition or list references easily [56]. Setting up an LSP (Language Server Protocol) for the project language in an editor also provides IDE-like navigation (jump to symbol, find all references).

- **Static Analysis Tools:** These can find relationships in code without running it. For example, **call graph generators** map which functions call which. Tools like *Doxygen* (for C/C++ and others) can produce caller/callee graphs if configured with GraphViz. For Python, a module like `pycallgraph` can visualize call relationships (though it runs the code to do so). Static analyzers like *clang tools* or *SonarQube* can also highlight complex functions, dependency cycles, etc., which hint at where tricky logic lies. Using a static analysis, you might generate an **include dependency graph** (which files include which, useful in C/C++) or a **package dependency graph** (which modules import which, useful in Java/Python) to see the coupling in the system.

- **UML and Architecture Diagrams:** Some projects include UML diagrams of their architecture in the docs [57]. These are gold for quickly grasping how components interact. If not provided, you can create your own high-level diagram as you learn. There are tools to assist: for instance, `pyreverse`

(part of Pylint) can auto-generate UML class diagrams for Python code [58]. Java IDEs often can export class diagrams. Even without fancy tools, you can sketch a block diagram: e.g., a box for "frontend" and "backend" connected by an API arrow, or a flowchart of the request handling path. This helps keep track of the big picture.

- **Visualizing Call Graphs:** A **call graph** is a directed graph of function calls at runtime. Visualizing one can be helpful to understand sequence of operations, especially for performance-sensitive code. For example, you might want to see what a `processTransaction()` function calls internally. There are automatic tools: for C/C++, Doxygen can emit a call graph image for each function; for Java, you might use an analysis tool or even runtime profilers. For Python, `pycallgraph` or using `cProfile` data with graph visualization (like gprof2dot) works. Below is an example of a call graph for a simple Python class, showing the flow when `check()` calls either `odd()` or `even()` based on the input number:



Example call graph generated for a Python program `WhatNumber` (functions are nodes and arrows indicate function calls). Such graphs help in understanding runtime execution paths; here, whenever `WhatNumber.check` is called, it may invoke either the `odd` or `even` method depending on the number's parity, incrementing a counter each time [59] [60]. Tools like pycallgraph (for Python) or Doxygen with GraphViz (for C/C++) can produce call graphs to visualize complex code flows.

Interpreting the graph: In the image, `WhatNumber.check` calls either `WhatNumber.odd` or `WhatNumber.even`. The graph might also annotate how many times each edge was taken if gathered from actual runs. In large programs, you wouldn't generate a full graph of everything (it would be overwhelming), but focusing on a particular module or sequence can clarify which functions are tightly coupled.

- **Dependency Visualizers:** Large projects have many external libraries. Tools such as `npm ls` for Node, `pipdeptree` for Python, or Maven's `dependency:tree` goal for Java can list all dependencies. There are also visualizers (like `pipenv graph` or third-party web tools) to show a

graph of dependencies and licenses. For internal dependencies (modules depending on modules), static analysis or build files are the source. Some IDEs have plugins to draw module relations. Understanding the dependency graph can be important for licensing (ensuring compatible licenses) and for spotting which components you need to rebuild or update when patching.

Patching and Modifying Open-Source Software: One of the great advantages of OSS is that you can modify it to suit your needs – whether to fix a bug, add a feature, or adjust behavior – and often contribute those changes back to the community.

- **Identifying Where to Patch:** Using the above techniques (search, reading, debug prints), pinpoint the code that needs change. For example, suppose an open-source library has a bug parsing a certain input. You might search the repository for that error message or function name you suspect is faulty. Reading related code and tests will narrow down the suspect location. The git history trick mentioned earlier can help find if that area was problematic (lots of recent edits or comments). Also, search the project's issue tracker; maybe someone has already identified the problematic code or a maintainer has hinted at the cause.

- **Making the Change Safely:** Set up a proper dev environment: fork the repo, create a new git branch for your patch, and get the project building. Make the code change and run the test suite to ensure nothing else broke. It's wise to add a new test case that reproduces the issue and verify it passes with your fix (this guards against regressions). Keep patches as focused as possible – one issue at a time – to ease review.

- **Submitting Contributions:** If you intend to contribute the patch upstream, follow the project's contribution guidelines. Typically, you would create a pull request on GitHub/GitLab with your changes and tests. Maintainers might require you to sign a Contributor License Agreement (CLA) or they might discuss tweaks to your solution. Even if it's a personal patch you don't plan to upstream, it's good practice to document it (in code comments or commit message) for your future self or colleagues.

- **Binary Patching:** Though in open source you have the luxury of source, sometimes one might distribute a small patch file or binary diff (e.g., if you want to temporarily monkey-patch a library in your project without forking it fully). Tools like `patch` (unified diff files) are the classic way to distribute source patches. For binary distribution, one could use `diff / xdelta` on compiled output, but that's less common in OSS context since source is available.

- **Legal/License Compliance:** Since the code is open-source, using and modifying it is permitted under its license, but **be mindful of license terms**. Open-source licenses generally allow you to *use, modify, and distribute* the code. However, if it's a copyleft license like the GPL, and you distribute a modified version (even internally to a client or as a product), you are obligated to release your source changes under the same license [61]. For more permissive licenses (MIT, Apache, BSD), you can usually distribute binaries without releasing source, but you must include the original copyright notice and attribution in documentation. When patching for your own use, you don't have to share the changes, but if you later share the binary or code with others, follow the license. In the UK/EU, remember that open-source licenses are legally binding. The European Court of Justice has even ruled that the *functionality* of a program is not protected by copyright (2012 case) and that creating compatible software is allowed [48]. But that doesn't mean you can ignore the license conditions on

distribution of code. In short: **internal analysis and modifications are fine**, and even encouraged in OSS; **external distribution requires compliance** (same license for derivatives if required, provide source or notices as needed). Fortunately, most OSS licenses explicitly permit making **derived works** and patching as long as the same rights are preserved in downstream copies [61].

Best Practices Recap: When approaching any large open-source project, break the task into steps. First, learn what the software is supposed to do (user perspective), then read its code with a goal in mind (don't try to understand everything – focus on the part relevant to your need) [62] [63]. Leverage tools to search and map the code. Gradually, as you tackle small issues or components, the bigger picture will become clearer – a concept known as the *"paper-cut principle"* (many small fixes across the codebase gradually give you broad understanding) [64]. Over time, you'll navigate the code faster and know where to look for what. Open-source communities are usually welcoming to reverse engineers and new contributors; you can often discuss in project forums or chat if you're stuck, as long as you've done due diligence. In the end, reverse engineering an open codebase is about reading and reasoning, using all available resources (documentation, runtime testing, static analysis) to build a mental model of the software. Once you have that, implementing a patch or new feature is much more feasible.

Conclusion: Reverse engineering is as much an art as a science – it combines technical know-how with investigative intuition. Whether disassembling a Mach-O binary to bypass an iOS jailbreak check, unraveling a packed webpack bundle to understand a web app's API calls, or combing through thousands of lines of open-source code to fix a bug, the core skills are similar: observation, pattern recognition, and iterative hypothesis testing (static analysis gives a hypothesis, dynamic test confirms or refutes it, and so on). Always choose the right tools for the job: a disassembler for native code, a debugger or instrumentation tool for runtime, a proxy for web traffic, or a static analyzer for large codebases. We also highlighted the importance of operating within legal allowances – the UK/EU frameworks provide room for reverse engineering in many cases (interoperability, security research, etc. [36]), but one must avoid crossing into misuse of proprietary code or violating licenses. By following a structured approach and referencing quality resources (many of which we cited throughout this guide), you can successfully reverse engineer software across these domains. Happy hacking (for the good cause)!

References:

- Corellium Blog – *Intro to iOS Mobile Reverse Engineering* [65] [10]
- Apriorit Article – *How to Reverse Engineer an iOS App* [11] [20]
- K. Sławiński, Paramount Tech – *Introduction to Reverse Engineering App Store Apps* [23] [29]
- Promon Blog – *iOS Obfuscation and Reverse Engineering* [28]
- React Masters (Medium) – *Extracting React Code from a Website* [41] [66]
- DreamFactory Blog – *How to Reverse Engineer APIs* [44]
- P. Singh – *General Guide for Exploring Large OSS Codebases* [67] [54]
- T. Mikulski (TME) – *Reverse engineering – what is it and is it legal?* [48] [36]
- Vidström Labs – *Legal boundaries of reverse engineering in the EU* [30] [34] (EU Directive excerpts)

---

[1] [2] [5] [9] [10] [21] [22] [65] iOS Reverse Engineering | Tools for Mobile Apps | Corellium
https://www.corellium.com/blog/ios-mobile-reverse-engineering

3 4 6 7 8 11 12 13 14 20 How to Reverse Engineer an iOS App - Apriorit
https://www.apriorit.com/dev-blog/how-to-reverse-engineer-an-ios-app

15 16 17 18 19 23 24 25 26 29 Introduction to reverse engineering of AppStore apps | Paramount Tech
https://paramount.tech/blog/2022/11/14/introduction-to-reverse-engineering-of-appstore-apps.html

27 [PDF] iOS Hacking Guide.pdf - Security Innovation
https://web.securityinnovation.com/hubfs/iOS%20Hacking%20Guide.pdf

28 Preventing reverse engineering with iOS code obfuscation | Promon
https://promon.io/security-news/ios-obfuscation

30 31 32 33 34 35 47 The legal boundaries of reverse engineering in the EU
https://vidstromlabs.com/blog/the-legal-boundaries-of-reverse-engineering-in-the-eu/

36 48 Reverse engineering - what is it and is it legal? | Transfer Multisort Elektronik
https://www.tme.eu/en/news/library-articles/page/56932/reverse-engineering-what-is-it-and-is-it-legal/

37 38 39 40 41 42 66 Process of Extract the React code from the website | by React Masters | Medium
https://medium.com/@reactmasters.in/process-of-extract-the-react-code-from-the-website-fc88763af0bd

43 MitmProxy2Swagger: Automagically reverse-engineer REST APIs
https://news.ycombinator.com/item?id=42572662

44 How to Reverse Engineer APIs: The Benefits and Tools - Blog
https://blog.dreamfactory.com/reverse-engineering-apis-the-benefits-and-tools

45 SPA source code recovery by un-Webpacking source maps - Medium
https://medium.com/@rarecoil/spa-source-code-recovery-by-un-webpacking-source-maps-ef830fc2351d

46 Restoring Frontend Source Code Using Sourcemaps - WellWells
https://wellstsai.com/en/post/restoring-source-code-from-sourcemaps/

49 50 51 53 54 55 56 57 58 59 60 62 63 64 67 General Guide For Exploring Large Open Source Codebases
https://pncnmnp.github.io/blogs/oss-guide.html

52 maintenance - How do you dive into large code bases? - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/6395/how-do-you-dive-into-large-code-bases

61 The Open Source Definition
https://opensource.org/osd