A Configurable Instrumentation Framework for Evaluating Android RASP Resiliency

I. Architecting the 'RaspEval' RASP Evaluation Framework

To effectively evaluate the robustness of a Runtime Application Self-Protection (RASP) solution, the evaluation framework itself must be designed to bypass the most common, low-level environmental checks that RASP solutions deploy. A test that is trivially detected provides no data on the RASP's more advanced defenses.
This analysis outlines an advanced evaluation framework, termed 'RaspEval', built upon the Frida instrumentation toolkit. The framework is designed to be configurable, autonomous, and capable of simulating a wide spectrum of attack vectors, from Java-layer hooking to direct memory patching.

Core Concept: Gadget-Based, Config-Driven Autonomous Instrumentation

The foundational architectural choice for 'RaspEval' is the use of Frida's **Gadget mode**, as opposed to its standard server-based "Injected" mode.[1]
This is a critical, non-negotiable design choice. A primary function of nearly all RASP solutions is to detect the presence of the frida-server binary by its process name or by scanning for its default listening port (27042).[1] An evaluation attempt using frida-server would be detected and terminated immediately, yielding no useful data about the RASP's hook-detection, anti-tracing, or memory-integrity capabilities.
The Frida Gadget (libgadget.so) is a shared library that is "embedded" within the target application's process space.[2] It is loaded by the Android linker just like any other native library (e.g., libnative-lib.so). This architecture offers two immediate advantages:

1. **Stealth:** It bypasses the most basic process- and port-scanning checks, as there is no separate server process or open network port to detect.
2. **Autonomy:** It runs within the app's context from the moment of launch, allowing for autonomous operation without a tethered host PC, thereby simulating a

repackaged-app attack vector.

By leveraging the Gadget, the 'RaspEval' framework bypasses this first line of defense, permitting the evaluation to proceed to the RASP's more sophisticated defensive layers.

The Configuration Engine: parameters and the init() Entry Point

The framework's configurability is powered by the Gadget's native JSON configuration system.[2] The deployment requires two files to be placed in the repackaged APK's lib/<abi>/ directory:

1. **The Gadget:** libgadget.so
2. **The Configuration File:** libgadget.config.so

The configuration file is a standard JSON file that has been renamed with a .so extension. This is a necessary workaround to satisfy the Android package manager, which typically only copies files prefixed with lib and suffixed with .so from the lib/ directory into the final installation path.[2]

This configuration file defines the main JavaScript payload and, most importantly, provides the attack configuration via the parameters key.[2]

Example libgadget.config.so:

JSON

```
{
  "interaction": {
    "type": "script",
    "path": "libscript.so",
    "on_change": "reload"
  },
  "parameters": {
    "java_hooks":,
        "action": "replace_return",
        "new_return_value": false
      }
    ],
    "native_hooks": [
      {
        "library": "libc.so",
        "function": "open",
        "action": "modify_arg",
        "arg_index": 0,
        "if_arg_contains": "/system/bin/su",
```

```
      "new_arg_value": "/dev/null"
    }
   ],
   "stalker_config": {
    "target_library": "librasp.so",
    "target_function": "run_tamper_checks"
   }
  }
}
```

The Gadget natively passes the JSON object specified in the parameters key directly to an init(stage, parameters) function within the main JavaScript file (libscript.so).[2]
This init function serves as the central nervous system for 'RaspEval'. It acts as a dispatcher, parsing the parameters object and activating the corresponding attack modules. All attack modules (Java, Native, Stalker) are defined as dormant functions or classes within the script, awaiting activation by this init dispatcher.
Example raspeval.js (renamed to libscript.so) Entry Point:

JavaScript

```
// Global config store
global.RASP_EVAL_CONFIG = {};

rpc.exports = {
   init(stage, parameters) {
      console.log(" Framework initializing. Stage: " + stage);
      global.RASP_EVAL_CONFIG = parameters;

      // Dispatch to modules based on config
      if (parameters.java_hooks) {
         JavaHookFactory.apply(parameters.java_hooks);
      }
      if (parameters.native_hooks) {
         NativeHookFactory.apply(parameters.native_hooks);
      }
      if (parameters.stalker_config) {
         StalkerModule.run(parameters.stalker_config);
      }
   }
};
```

This architecture fully satisfies the requirement for a configurable, JSON-driven evaluation

framework.

II. Module 1: The Java/Kotlin Instrumentation Engine

This module details the techniques for instrumenting the Java/Kotlin runtime. It addresses critical challenges in timing, concurrency, and obfuscation that are common when evaluating RASP-protected applications.

Hooking Application Entry Points (The Timing Problem)

A fundamental RASP evaluation test is to hook the application's earliest entry points, such as android.app.Application.onCreate [5] or android.app.Activity.onCreate.[6] These methods often initialize the RASP defenses.
Attempting to hook these methods by *attaching* (frida -U -n <pkg>) will fail, as the methods will have already executed by the time Frida attaches.[9]
The solution is to *spawn* the application (frida -U -f <pkg>).[10] However, a race condition often still exists: the script loading may not complete before Application.onCreate is called.
The only 100% reliable method to win this race condition is the **manual spawn-load-resume sequence**, which must be implemented in a host script (e.g., Python):

1. pid = device.spawn(['com.target.pkg']): This spawns the application process, but its main thread is automatically *suspended* by default.
2. session = device.attach(pid): Frida attaches to this suspended process.
3. script = session.create_script(jscode)
4. script.load(): Frida injects the script, and all hooks (e.g., for Application.onCreate) are placed into memory. The app is *still suspended*.
5. device.resume(pid): The host script *now* resumes the main thread. The thread immediately begins execution and runs directly into the pre-placed hooks.[5]

This sequence is essential for testing a RASP's initialization logic.

Intercepting Asynchronous and UI-Bound Logic

RASP checks are frequently deferred into asynchronous java.lang.Runnable tasks to be run later. Furthermore, a 'RaspEval' test may need to simulate UI interaction (e.g., clicking a button) to trigger a RASP-protected feature.
This presents a major challenge: Android's concurrency rules forbid UI modification from any thread other than the main UI thread. A standard Frida hook (Java.use...) executes on a dedicated Frida thread, and any attempt to call a UI function (like TextView.setText()) from it will crash the application.
The solution is a "UI-Thread Bridge Attack," which uses Frida's Java.registerClass API to

dynamically create a new Java class *at runtime* [13]:

1. A new class is defined in JavaScript that implements:.
2. The run() method of this new class contains the UI-modifying code (e.g., instance.onClick(view)).
3. The original hook (on the Frida thread) obtains an instance of the current Activity (e.g., via Java.choose).
4. The hook calls activity.runOnUiThread(MyNewRunnable.$new()).

This technique schedules the malicious code to run on the correct UI thread, bypassing the crash and allowing the framework to simulate complex user interactions to trigger protected workflows.

This module also supports manipulation of method arguments, such as the android.os.Bundle object passed to onCreate.[6] Hooks can be configured to read or inject data into this Bundle to simulate different application launch states.

Example: Injecting data into onCreate Bundle based on JSON config

JavaScript

```
Java.perform(() => {
    const Activity = Java.use("com.target.app.MainActivity");
    const String = Java.use("java.lang.String");

    Activity.onCreate.overload("android.os.Bundle").implementation = function(bundle) {
        if (global.RASP_EVAL_CONFIG.inject_bundle) {
            let key = global.RASP_EVAL_CONFIG.bundle_key;
            let val = global.RASP_EVAL_CONFIG.bundle_val;

            console.log(`[Java.Inject] Injecting into Bundle: ${key} -> ${val}`);
            // Create new Java String objects for the call
            let javaKey = String.$new(key);
            let javaVal = String.$new(val);

            // Call the Bundle.putString() method
            bundle.putString(javaKey, javaVal);
        }

        // Call the original onCreate
        return this.onCreate.overload("android.os.Bundle").call(this, bundle);
    };
});
```

Dynamic Hooking and Obfuscation Resilience

RASP-protected apps are invariably obfuscated. The 'RaspEval' framework must be resilient to two common challenges:

1. **Method Overloads:** Standard SDK methods and obfuscated methods often have multiple overloads. A hook attempt that fails to specify the *exact* signature will fail.[14] The framework's JSON config must accept an array of signatures (e.g., ``) [15], and the hook factory must use the .overload() API.[16]
2. **Field/Method Naming Conflicts:** Obfuscators (like R8/ProGuard) create name collisions, such as a field named a and a method named a(). A hook attempting to access this.a will ambiguously target the method, not the field. Frida provides a non-obvious syntax to resolve this:
    - this.a(): Accesses the **method**.
    - this._a: Accesses the **field** object.
    - this._a.value: Accesses or sets the **value** of the field.[16]

A RASP bypass (e.g., this.is_rooted.value = false) must use this ._field.value syntax. The 'RaspEval' hook factory must be configured to use this setter when a hook's action is defined as replace_field instead of replace_implementation.

III. Module 2: The Native Attack Simulation Engine

Many RASP solutions place their most sensitive checks in native C/C++ libraries (.so files) to evade Java-level instrumentation. This module simulates attacks against this native layer.

Native Function Interception (Interceptor.attach)

The core API for native hooking is Interceptor.attach(target, callbacks).[20] The target address is not hardcoded; it is dynamically located at runtime using Module.findExportByName(library_name, function_name).[20] The library_name (e.g., "libc.so") and function_name (e.g., "open") are supplied by the JSON config.

Advanced Native Argument Manipulation (Simulating I/O Redirection)

A prime target for this module is libc.so's open or fopen function.[24] RASP solutions often hook these functions themselves to monitor for filesystem access, particularly reads of root-indicating files like /system/bin/su.[26]
'RaspEval' tests this defense by performing a "double hook." It also hooks open and modifies

the path argument. This tests the RASP's hook integrity: does it see the *original* path, or the *modified* one?

Implementation:

1. **onEnter(args):** This callback receives an array of NativePointer objects representing the arguments.[20] For open(const char *path,...), args is the pointer to the path string.
2. **Read Path:** The original path is read using args.readUtf8String().[20]
3. **Conditional Modification:** Based on the JSON config (e.g., if_arg_contains: "/system/bin/su"), the pointer is replaced.

The critical distinction is that the framework does not modify the string at the original memory location. Doing so is unreliable and may crash the app. Instead, it allocates *new* memory for the new path (e.g., "/dev/null") and *replaces the pointer* in the args array.[27]

Example: libc.so open hook

JavaScript

```
Interceptor.attach(Module.findExportByName("libc.so", "open"), {
  onEnter(args) {
    const path = args.readUtf8String();

    // Check against config
    if (path && path.includes("/system/bin/su")) {
      console.log(`[Native.Hook] Intercepted open() for: ${path}`);

      // Allocate new memory and replace the pointer in args
      const newPath = Memory.allocUtf8String("/dev/null");
      args = newPath;

      console.log(`[Native.Patch] Replaced path with: /dev/null`);
    }
  }
});
```

Spoofing Native Function Results (retval.replace)

This technique is used to fake the results of native environment checks, such as a RASP's internal check_tampering() function that might return 1 if tampering is detected.

Implementation:

The onLeave(retval) callback is used. The retval object is a NativePointer-derived object containing the raw return value.20

This value can be forcibly overwritten. For example, retval.replace(ptr("0x0")) will force the function to return 0 (success) to its caller, regardless of the original computation.20
Example: Forcing a RASP check to return "success"

JavaScript

```
Interceptor.attach(Module.findExportByName("librasp.so", "check_tampering"), {
  onEnter(args) {
    console.log("[Native.Hook] librasp!check_tampering called");
  },
  onLeave(retval) {
    console.log(`[Native.Hook] Original return value: ${retval}`);

    // Force return '0' (success/not-tampered)
    retval.replace(ptr("0x0"));

    console.log("[Native.Patch] Forced return value to 0");
  }
});
```

IV. Module 3: Direct Memory Attack and Tracing

This module simulates the most aggressive attack vectors: direct modification of the process's code and memory, and deep execution tracing. These are designed to test a RASP's highest-level defenses, such as runtime memory integrity scanning and anti-tracing.

Simulating Memory Patching Attacks (Memory.patchCode)

This test simulates an in-memory patch, such as overwriting a function's prologue with a JMP hook or, more simply, forcing it to return a specific value. This directly tests the RASP's ability to detect modification of its own code segments.
A critical, non-obvious prerequisite exists for this attack. A process's code segment (.text) is mapped into memory as read-execute (r-x). A direct call to Memory.patchCode on this region will fail with a segmentation fault.[31]
The solution requires a two-step process:

1. **Memory.protect(targetPtr, pageSize, 'rwx'):** The memory page containing the target function must first be made *writable*.[32]
2. **Memory.patchCode(...):** The patch can now be applied.

This two-step process creates a twofold test for the RASP:

1. Does the RASP detect the suspicious mprotect system call on its own code segment? A robust RASP should.
2. If the mprotect call succeeds, does the RASP *then* detect the code modification, for example via a runtime checksum of its .text section?

Example: Patching a native function to return 1337

JavaScript

```
const funcPtr = Module.findExportByName("librasp.so", "is_secure");

// 1. Make the memory page writable
Memory.protect(funcPtr, 4096, 'rwx');
console.log("[Memory.Patch] Changed page to RWX");

// 2. Patch the code
Memory.patchCode(funcPtr, 16, (code) => {
   const cw = new Arm64Writer(code, { pc: funcPtr });

   // Injects:
   // ldr x0, #1337 (or equivalent mov)
   // ret
   cw.putLdrRegU64('x0', 1337);
   cw.putRet();
   cw.flush();
});
console.log("[Memory.Patch] Patched is_secure() to return 1337");
```

This example uses Arm64Writer to generate AArch64 machine code, replacing the function with a stub that loads 1337 into the return register (x0) and immediately returns.[31]

Memory Reconnaissance and Patching (Memory.scan)

This module simulates a full attack chain: reconnaissance followed by patching. It tests if the RASP detects either the aggressive memory scan or the subsequent write.

The Memory.scan(address, size, pattern, callbacks) API is used to search a memory range for a specific byte pattern.[20] The pattern is a hex string (e.g., "7f 45 4c 46" for an ELF header [35]) and supports wildcards (??).[31]

The JSON config provides a search pattern and a patch pattern. The onMatch callback then writes the patch bytes directly to the found address using address.writeByteArray().

Example: Scanning for an API key and replacing it

JavaScript

```
// Config from global.RASP_EVAL_CONFIG.scan_patch
const config = {
   "module_name": "libconfig.so",
   "pattern": "41 50 49 5f 4B 45 59 5f 48 45 52 45", // "API_KEY_HERE"
   "patch_pattern": "46 41 4B 45 5f 4B 45 59 5f 48 45 52 45"  // "FAKE_KEY_HERE"
};

const m = Process.getModuleByName(config.module_name);

Memory.scan(m.base, m.size, config.pattern, {
   onMatch(address, size) {
      console.log(` Pattern found at ${address}`);

      // Convert hex string "00 11 22" to byte array [0x00, 0x11, 0x22]
      const patchBytes = config.patch_pattern.split(' ')
        .map(hex => parseInt(hex, 16));

      // Patch in-place
      address.writeByteArray(patchBytes);
      console.log(`[Memory.Patch] Patched ${patchBytes.length} bytes at ${address}`);

      return 'stop'; // Stop scanning after first match
   },
   onComplete() {
      console.log(" Scan complete.");
   }
});
```

Instruction-Level Tracing with Stalker

This is the most advanced test, designed to defeat RASP anti-tracing defenses. RASP solutions may detect high-level Interceptor hooks by checking function prologues or stack traces.[1]
Stalker, however, is a dynamic recompilation (JIT) engine. It copies basic blocks of instructions, instruments them, and executes the copy. It is far stealthier, and many RASP

checks are not effective against it.[36]
The 'RaspEval' strategy is to hook a high-level RASP function (e.g., run_checks) with Interceptor and, in the onEnter callback, begin *stalking* the current thread: Stalker.follow(this.threadId,...).[37]
The transform callback is the core of this attack.[38] It iterates over every instruction in a basic block *before* it is executed. iterator.putCallout(callback) is used to insert a call to our JavaScript code at any instruction.[37] This callback receives the full CPU context object, providing access to all registers (context.pc, context.x0, context.sp, etc.).[40]
This provides a complete, instruction-by-instruction disassembly of the RASP's internal logic *as it executes*, rendering all static analysis and obfuscation useless.
Example: Tracing a RASP check function at the instruction level

JavaScript

```
Interceptor.attach(Module.findExportByName("librasp.so", "run_checks"), {
  onEnter(args) {
    console.log(" Following thread " + this.threadId);

    Stalker.follow(this.threadId, {
      transform: function(iterator) {
        let instruction;
        const moduleBase = Process.getModuleByName("librasp.so").base;

        while ((instruction = iterator.next())!== null) {
          // Only trace instructions inside the RASP library
          if (instruction.address.compare(moduleBase) >= 0) {

            // Insert a callout at *every* instruction
            iterator.putCallout((context) => {
              const offset = context.pc.sub(moduleBase);
              console.log(` 0x${offset.toString(16)}: ${instruction}`);
            });
          }
          iterator.keep(); // Keep the original instruction
        }
      }
    });
  },
  onLeave(retval) {
    Stalker.unfollow(this.threadId);
    console.log(" Unfollowed thread " + this.threadId);
```

```
    }
});
```

V. Framework Integration and Deployment

This section synthesizes all modules into the final, deployable 'RaspEval' package.

The Configurable Gadget Package

The final deployment process involves:

1. Decompiling the target APK using a tool like apktool.[41]
2. Adding the Frida Gadget, libgadget.so, to the lib/<abi>/ directory.[2]
3. Creating the master JSON config file and adding it to the *same* directory as libgadget.config.so.[2]
4. Creating the main raspeval.js script (containing all logic from Modules I-IV), renaming it to libscript.so to ensure the package manager includes it, and placing it in the same lib/<abi>/ directory.[4]
5. Recompiling and re-signing the modified APK.

When the user launches this repackaged app, the Android linker loads libgadget.so, which reads libgadget.config.so, which in turn loads libscript.so and passes it the parameters JSON object, initiating the autonomous evaluation.

The 'RaspEval' init Function (The Dispatcher)

The following code is the complete entry point for the raspeval.js script. It demonstrates how the parameters object, passed from the Gadget [2], is used to dynamically configure and launch the attack simulations.

JavaScript

```
// This 'global' will hold our config for other modules
global.RASP_EVAL_CONFIG = {};

/*
 * DEFINE MODULES HERE
 * (e.g., JavaHookFactory, NativeHookFactory, StalkerModule, etc.)
 */
```

```
const JavaHookFactory = {
    apply: function(hooks) {
        console.log(`[JavaHookFactory] Applying ${hooks.length} Java hooks...`);
        //... (Logic to iterate hooks and apply Java.use)
    }
};
const NativeHookFactory = {
    apply: function(hooks) {
        console.log(`[NativeHookFactory] Applying ${hooks.length} native hooks...`);
        //... (Logic to iterate hooks and apply Interceptor.attach)
    }
};
const StalkerModule = {
    run: function(config) {
        console.log(` Initializing Stalker for ${config.target_function}`);
        //... (Logic to set up Stalker hook)
    }
};
//... etc.

// This is the main entry point called by the Gadget
rpc.exports = {
    init(stage, parameters) {
        console.log(" Framework initializing...");
        console.log(" Stage: " + stage);
        console.log(" Config: " + JSON.stringify(parameters));

        // Store config for other modules to access
        global.RASP_EVAL_CONFIG = parameters;

        // Use Java.perform to ensure Java VM is ready
        Java.perform(() => {
            // Dispatch to modules based on config
            if (parameters.java_hooks && parameters.java_hooks.length > 0) {
                JavaHookFactory.apply(parameters.java_hooks);
            }

            if (parameters.native_hooks && parameters.native_hooks.length > 0) {
                NativeHookFactory.apply(parameters.native_hooks);
            }

            if (parameters.stalker_config) {
                StalkerModule.run(parameters.stalker_config);
```

```
        }
      });

      console.log(" Framework initialization complete.");
    }
};
```

VI. Interpreting 'RaspEval' Results: A RASP Resiliency Matrix

The 'RaspEval' framework provides a direct, empirical method for testing the specific claims made by RASP solution vendors.[43] Claims of "hook detection" [46], "tamper prevention" [1], and "anti-injection" [47] can be mapped 1:1 to the simulation modules of this framework.
The output of the evaluation is the app's behavior: it either crashes/exits (indicating a successful detection by the RASP) or it continues to run with the hook/patch in place (indicating a failure of the RASP defense). This binary outcome allows for the creation of a clear RASP Resiliency Matrix.

RASP Resiliency Matrix

RASP Defense Vector (Vendor Claim)	'RaspEval' Simulation Module / Technique	Test Goal	Pass / Fail Criteria
Environment/Tooling Detection	**Framework:** frida-gadget loaded [2]	Does RASP detect the frida-gadget.so by name or signature? [1]	**Pass:** App crashes or exits, citing "Frida" or "tampering." **Fail:** App runs, init function logs.
Java-Layer Hook Detection	**Module I:** Java.use() [8]	Does RASP detect hooks on Java/Kotlin methods?	Pass: App crashes/exits on hook application. Fail: Hook is successful, logs "onClick".19
Native-Layer Hook Detection	**Module II:** Interceptor.attach() [20]	Does RASP detect hooks on native functions (e.g., in libc.so or its own .so)?	Pass: App crashes/exits on hook application. Fail: Hook is

			successful, logs "open".25
I/O Tampering Detection	**Module II:** open hook with args replacement [28]	Does RASP detect that a file path argument was modified in-flight?	**Pass:** App detects the *original* path (e.g., "/system/bin/su") despite the hook. **Fail:** RASP sees the *modified* path (e.g., "/dev/null").
Code Integrity / Memory Tampering	**Module III:** Memory.patchCode() on a function [31]	Does RASP perform runtime integrity checks/checksums on its code segments?	**Pass:** App crashes/exits after patch, citing "memory tampering." **Fail:** App runs, and patched function returns 1337.
Memory Write Detection	**Module III:** Memory.protect('rwx') [33]	Does RASP detect the mprotect syscall used to make code writable?	**Pass:** App crashes/exits immediately on the Memory.protect call. **Fail:** The Memory.protect call succeeds silently.
Memory Scanning Detection	**Module III:** Memory.scan() [20]	Does RASP detect its own memory being scanned for patterns?	**Pass:** App crashes/exits during the scan. **Fail:** Scan completes and finds the pattern.
Execution Tracing Detection	**Module IV:** Stalker.follow() [37]	Does RASP detect JIT-based instruction tracing? [36]	**Pass:** App crashes/exits, or Stalker fails to follow. **Fail:** Stalker produces a full instruction trace of the RASP check.

Works cited

1. Frida Detection & Prevention - Approov, accessed October 27, 2025, https://approov.io/knowledge/frida-detection-prevention
2. Gadget | Frida • A world-class dynamic instrumentation toolkit, accessed October 27, 2025, https://frida.re/docs/gadget/
3.
4. 09 - How to use frida on a non-rooted device — LIEF Documentation, accessed October 27, 2025, https://lief.re/doc/latest/tutorials/09_frida_lief.html
5. Can't hook Application.onCreate ? · Issue #608 · frida/frida - GitHub, accessed

October 27, 2025, https://github.com/frida/frida/issues/608
6. MK/hacktricks - Gitee, accessed October 27, 2025, https://gitee.com/scriptkiddies/hacktricks/blob/master/mobile-apps-pentesting/android-app-pentesting/frida-tutorial/README.md
7. unable to hook onCreate() using frida for android application - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/73855422/unable-to-hook-oncreate-using-frida-for-android-application
8. Hooking & Intercepting | DroidTomeSec - Android Pentesting - GitBook, accessed October 27, 2025, https://e1mazahy.gitbook.io/droidtomesec/android-pentesting/frida/hooking-and-intercepting
9. Frida not hooking function in private process - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/64211144/frida-not-hooking-function-in-private-process
10. Frida basics - Frida HandBook, accessed October 27, 2025, https://learnfrida.info/basic_usage/
11. How to hook Android Native methods with Frida (Noob Friendly) | - erev0s.com, accessed October 27, 2025, https://erev0s.com/blog/how-hook-android-native-methods-frida-noob-friendly/
12. how to hook onCreate medthod in Application Class · Issue #480 · frida/frida - GitHub, accessed October 27, 2025, https://github.com/frida/frida/issues/480
13. Calling an API to modify an App's GUI from non-Main thread in Frida - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/65790594/calling-an-api-to-modify-an-apps-gui-from-non-main-thread-in-frida
14. Android Instrumentation - Frida HandBook, accessed October 27, 2025, https://learnfrida.info/java/
15. How to overload function with "int" in Frida - android - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/57553420/how-to-overload-function-with-int-in-frida
16. Overcoming Some "Gotcha's" in Frida | rastating.github.io, accessed October 27, 2025, https://rastating.github.io/overcoming-some-gotchas-in-frida/
17. Hook all overloads - Java/Android - Frida - GitHub Gist, accessed October 27, 2025, https://gist.github.com/FrankSpierings/6e2608e22121b1aeaaa4588f13387dde
18. Android Hooking in Frida | Node Security, accessed October 27, 2025, https://node-security.com/posts/android-hooking-in-frida/
19. Android | Frida • A world-class dynamic instrumentation toolkit, accessed October 27, 2025, https://frida.re/docs/examples/android/
20. JavaScript API | Frida • A world-class dynamic instrumentation toolkit, accessed October 27, 2025, https://frida.re/docs/javascript-api/
21. Exploring Native Functions with Frida on Android — part 3 | by The Mobile Security

Guys, accessed October 27, 2025, https://mobsecguys.medium.com/exploring-native-functions-with-frida-on-android-part-3-45422ae18caa
22. Security Notes #002: Getting Started with Frida | by pwniel - Medium, accessed October 27, 2025, https://medium.com/@daniel.bergers/security-notes-002-getting-started-with-frida-4f1429670998
23. Exploration of Native Modules on Android with Frida - Payatu, accessed October 27, 2025, https://payatu.com/blog/exploration-of-native-modules-on-android-with-frida/
24. How to hook fopen using Frida in Android? - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/56547020/how-to-hook-fopen-using-frida-in-android
25. Frida cheat sheet - Home - Awakened, accessed October 27, 2025, https://awakened1712.github.io/hacking/hacking-frida/
26. Using Frida to Find Hooks | Corellium Support Center, accessed October 27, 2025, https://support.corellium.com/features/frida/using-frida-find-hooks
27. Frida Dynamic instrumentation to bypass Root detections Part II | by Musyoka Ian - Medium, accessed October 27, 2025, https://musyokaian.medium.com/frida-dynamic-instrumentation-to-bypass-root-detections-part-ii-b7890dadfdcb
28. Functions | Frida • A world-class dynamic instrumentation toolkit, accessed October 27, 2025, https://frida.re/docs/functions/
29. How to modify return String value when hook native in Android · Issue #449 - GitHub, accessed October 27, 2025, https://github.com/frida/frida/issues/449
30. Read value from frida hooked native method basic_string parameter - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/68243063/read-value-from-frida-hooked-native-method-basic-string-parameter
31. Advanced Frida - Frida HandBook, accessed October 27, 2025, https://learnfrida.info/advanced_usage/
32. Frida Advanced Usage Part 8 – Frida Memory Operations Continued - 8kSec, accessed October 27, 2025, https://8ksec.io/frida-advanced-usage-part-8-frida-memory-operations-continued/
33. Memory scanning in Frida / Frida and Flutter apps | by Joel Teo - Medium, accessed October 27, 2025, https://medium.com/@orangecola3/memory-scanning-in-frida-frida-and-flutter-apps-3c1f4f6cc6ca
34. Shellcoding an Arm64 In-Memory Reverse TCP Shell with Frida - VerSprite, accessed October 27, 2025, https://versprite.com/vs-labs/frida-engage-part-two-shellcoding-an-arm64-in-memory-reverse-tcp-shell-with-frida/
35. Advanced Frida Usage Part 7 – Frida Memory Operations - 8kSec, accessed

October 27, 2025, https://8ksec.io/advanced-frida-usage-part-7-frida-memory-operations/
36. iOS Anti-Reversing Defenses, accessed October 27, 2025, https://pentest.y-security.de/Mobile%20Security%20Testing%20Guide/2023/0x06j-Testing-Resiliency-Against-Reverse-Engineering/
37. Stalker | Frida • A world-class dynamic instrumentation toolkit, accessed October 27, 2025, https://frida.re/docs/stalker/
38. Advanced Frida Usage Part 10 – Instruction Tracing using Frida ..., accessed October 27, 2025, https://8ksec.io/advanced-frida-usage-part-10-instruction-tracing-using-frida-stalker/
39. DBI Frida Stalker is not tracing instructions - Stack Overflow, accessed October 27, 2025, https://stackoverflow.com/questions/69815481/dbi-frida-stalker-is-not-tracing-instructions
40. Frida Stalker - Tracing binary instructions - YouTube, accessed October 27, 2025, https://www.youtube.com/watch?v=BgICyi2H2CU
41. Android Root Detection Bypass (Frida Hooking and APK Patching) - YouTube, accessed October 27, 2025, https://www.youtube.com/watch?v=JDWO3pVOOLc
42. Gadget Configurations · sensepost/objection Wiki - GitHub, accessed October 27, 2025, https://github.com/sensepost/objection/wiki/Gadget-Configurations
43. A Guide to Runtime Application Self-Protection (RASP) - Imperva, accessed October 27, 2025, https://www.imperva.com/resources/whitepapers/Imperva-A-Guide-to-RASP.pdf
44. Runtime Application Self-Protection (RASP), Investigation of the Effectiveness of a RASP Solution in Protecting Known Vulnerable Target Applications - SANS Institute, accessed October 27, 2025, https://www.sans.org/white-papers/38950
45. A Methodology Guide to Bypassing RASP and Root Detection in ..., accessed October 27, 2025, https://medium.com/@abhijithknamboothiri96/a-methodology-guide-to-bypassing-rasp-and-root-detection-in-mobile-apps-with-frida-17e4f97b88bc
46. White Paper — Bugsmirror Defender: Cutting-Edge Runtime Application Self-Protection (RASP) for Mobile Apps - Medium, accessed October 27, 2025, https://medium.com/@bugsmirror/white-paper-bugsmirror-defender-cutting-edge-runtime-application-self-protection-rasp-for-4de6e6a57d2e
47. Appdome Anti-Frida Solution Guide © 2024, accessed October 27, 2025, https://www.appdome.com/wp-content/uploads/2024/05/Appdome-Anti-Frida-Solution-Guide-%C2%A9-2024.pdf