All Your BASE Are Belong To You: Improved Browser Anonymity and Security on Android

Author: Peter Saisi
Advisor: Charles C Palmer
Dartmouth College

Dartmouth Computer Science Technical Report TR2016-800

Abstract

Android is the most popular [1] mobile operating system in the world. Android holds a marketshare of 82% with iOS, its nearest rival, managing a distant 13.9%. Android's unparalleled ubiquity makes it a popular target for malware and malvertising. Specifically, Android browsers have been targeted because many users spend great durations of time browsing the Internet. Unfortunately, as ways to track, fingerprint, and exploit unsuspecting users have increased, **B**rowsing **A**nonymity and **Se**curity (BASE) has contrastingly stalled. Third party apps seeking to displace the oft-maligned stock browser tend to focus on user privacy and defer malware defense to default operating system protections. This thesis introduces a novel browser - Congo. Congo's recursive definition, Congo's Obeism Negates Gentile Occurrences, hints at an augmented browser with a hardened sandbox(malware deterrent) and reinforced privacy protection (malvertising deterrent). Importantly, Congo requires no kernel modification thus making it readily available to Android OS versions later than Froyo. A reference mechanism, by the name Kinshasa, underpins the integrity and security of Congo.

1 Introduction

BASE is not an issue solely restricted to individual users. The explosion of mobile devices has not spared corporate institutions. In 2015, 80% [2] of global businesses self-reported that mobile phones were a medium-to-high security priority. Despite the fact that some businesses may mitigate exposure by providing locked down devices, about 59% of businesses employ a BYOD policy. Since 2011, there have been about 70 [3] reported Android botnets discovered. Contrastingly, iOS has had only 2 [4] recorded botnets. This huge difference in fates is partly because of Android's Bazaar [5] ethos. Android espouses an open constitution that undergirds user freedom, but unfortunately also exposes itself to attack through the very channel of openness. The emergence of bots highlights the gradual, but expected, commercialization of Android malware.

Contribution: In this thesis, Congo, a hardened Android mobile browser is introduced. Congo is sandboxed by the OS similar to any other Android app. However, unlike most Android apps, Congo is further restricted within the default sandbox by another bespoke sandbox enforced by Kinshasa. The extra sandbox employs the principle of least privilege. Consequently, Congo has bare and time-constrained privileges. The sandbox is achieved through sideloading a custom library that injects and hooks into Congo's *Procedure Linking Table*. Library injection and hooking is not particularly novel on Android. However, what makes Congo's approach interesting is that it:

(a) Works on all Android versions later than Froyo
(b) Requires no rooting or kernel modification
(c) Requires no separate application for ancillary work such as parsing Android Parcels
(d) Implements a refined means of intercepting binder communications and parsing Parcels

At the time of writing, no available formal research on application sandboxing achieves all the enumerated above propositions. Interestingly, some of the injection tactics used are reminiscent of persistent Android malware [6]. Dynamic instrumentation, linking and loading of executables, and Android overloaded interprocess communication are all leveraged to create a reference monitoring mechanism that invigilates application activity. Availability of Android source code provided great insight into the ARM linker [7] as implemented through the aptly named *crazy_linker* [8] suite of files. Given how resource constrained mobile devices are, Congo's overhead is also a point of focus. Congo's

main advantage over other mobile browsers is that it not only minimizes a user's fingerprint, but also proactively guards against malware. The Congo sandbox's edge over other implementations is its undemanding installation i.e., a person need only download and install the APK with no need to resort to rooting or flashing device firmware.

2 Background on Android

2.1 Introduction to Android

Android is an open source operating system [9]. Its stack comprises

- Android Kernel - a patched version of the mainstream Linux kernel (often with the same version number). It is the core of the operating system and provides fundamental low-level services.
- Middleware - comprises application layer (Frameworks i.e. Media Framework) and native libraries (i.e. OpenSSL). This layer provides the core of Application APIs available to Android applications
- Application layer - comprises both stock i.e. Clock and third party applications i.e. Firefox Browser for Android

1: Android Stack

2.2 Android Execution Model

Android applications are packaged using the eponymous APK(Android application package) file format. An APK is an archive that contains resources needed for execution of an application. Most importantly, it contains files in dex. Dex files, platform independent executable bytecode, are the principle entry point for execution. In Android versions younger than 4.4, the Dalvik runtime performs JIT(Just-In-Time) compilation of dex bytecode. Android 4.4, introduced ART(Android Runtime). In its contemporary form, ART performs AOT(Ahead-of-Time) compilation that converts Dalvik bytecode into system dependent binary. AOT's advantage is that an app is pre-compiled only once (when it is installed). JIT does repeated and continual interpretation of the app every time it runs. Consequently, JIT has a bigger memory footprint because the app and JIT interpreter run concurrently. AOT does not have such memory pressure or latency because it is compile-once and executes natively thereafter. AOT's reduced footprint leads to reduced power consumption in portable devices.

2.3 Android Application Security Model

2.3.1 Installation

Android enforces the requirement that all apps are digitally signed. This policy accommodates self-generated certificates by developers. When the OS is about to update an app, the update proceeds only if the proposed update's certificate matches the current app's certificate.

Android versions after 2.3 include an opt-in *Verify Apps* feature. This feature was first introduced for v4.2, but later backported to earlier versions. *Verify Apps* scans an app at install time for malicious signatures. Verification is done mainly with a view to protect users from malicious app sideloading. App sideloading, in this context, is the installation of apps by channels other than the officially sanctioned Google Play Store.

2.3.2 Execution

All apps and services run in individual sandboxes. Consequently, system services are sandboxed, albeit with elevated privileges. Outside the kernel ring, however, there is no provision for root or superuser with unconstrained access. The virtual sandbox limits what the application can access outside itself. Nevertheless, some applications usually do require external resources. For instance, a Phone Dialer application would need to access `android.telephony` service. Android solves this by instituting a permissions model. Whenever a user installs an app, Android reads the app manifest to determine which permissions the app has declared it needs. When the app is launched, Android asks the user to approve granting permission(s) to the app. A benefit of this is that the user can easily adjudge an app's nefariousness i.e. a Siren app, dedicated to only making siren sounds, would be out of place asking for SMS permissions.

Android's process privileging and isolation is a departure from Linux. Each app is assigned a unique user ID (UID) and run as the user in a separate process. Linux contrastingly has multiple applications running with same user permissions. The Android kernel enforces process isolation through standard means of user and group IDs. Google defines the sandbox as "simple, auditable, and based on decades-old, UNIX-style user separation of processes and file permissions" [10]. Discretionary Access Controls, such as file permissions, ensure that an `application A`'s private data directory is not accessible by an `application B`. Additionally, Android employs Security Enhanced Linux (SELinux) to ensure Mandatory Access Control over all processes, even processes running as root and superuser. SELinux on Android enforces restrictions and logs any violations that may occur. The kernel-level sandbox has the implication that all the layers above it (middleware, application framework) run within the application sandbox. Consequently, native code is as secure as interpreted code. Additionally, C code written via the NDK(Native Development Kit) can be just as secure as vanilla Java code. Any local memory corruption only affords privilege within the app's sandbox.

Android 4.1 (Jelly Bean) introduced a new `android:isolatedProcess` attribute for services. This attribute allows for creation of a process entirely separate from the parent with no privileges of its own. Such a feature is particularly useful for creating unprivileged independent processes dedicated to handling untrusted data such as JavaScript or PDF files.

The primary sanctioned manner by which applications can communicate outside the sandbox is through Android's interprocess communication channel called Binder. A client wishing to use a service cannot communicate directly with the process providing the service. Binder acts as the middleperson that mediates such requests. A client process issues a request, containing details such as method to execute and arguments, to the binder. The binder adds extra information, such as the client's UID, to the request. Thereafter, the transaction is relayed to the service. Addition of the UID is necessary as it acts as a unique identifier. When a process requests a service; the reference monitor ensures the process indeed has relevant permissions. For instance, `application A` with only telephony permissions making a request to the location service will encounter an insufficient permissions exception.

3 Requirements Analysis

The main objectives of `Congo`'s framework are:

- **Version agnosticism:** The sandbox works on versions of Android later than 2.3. Code accounting for platform specific differences is fewer than 20 lines. Most of the platform lines account for changes in API and resource locations in the Android source
- **No source modification:** The sandbox requires not root privilege nor does it require modification of kernel. Resultantly, it is portable across the most customized of Android images.
- **Context aware:** Unlike predominant security models that are context agnostic, the reference monitor enforces policies with consideration of context parameters such as time and location.
- **Light coupling:** The sandbox is simple and generic. It easily couples with existing code in no more than 10 lines of code for basic integration. Other methods would require rooting and installation of a third party framework such as `xposed`.

3.1 Threat Model

For full coverage, it is assumed that the host device:

- Is on stock Android
- Is not rooted

The proposition that stock and unrooted Android is required for full coverage does not imply `Congo` is inoperable on modified devices. Instead, the implication is that rooting or flashing Android will reduce `Congo`'s coverage. Rooting introduces glaring exploitable avenues, such as activation of superuser beyond the kernel, in the underlying OS. Such avenues inductively weaken `Congo`'s guarantee of security.

There are two assumed kinds of attackers :

- **Trackers:** Characteristic of advertisers who aggregate device and behavioral information to create attributable fingerprints of devices. Device information includes media access control (MAC) address, screen resolution, or browser user agent. Behavioral information includes typing rate, scroll rate, or browsing history.
- **Crackers:** Characteristic of malicious actors who attempt code execution on target device. For instance, a maladvert may load malicious JavaScript that attempts Java method invocation. `Kinshasa` works predominantly to mitigate this threat class.

3.2 Related Work

3.2.1 Tor

Tor-oriented browsers such as Orfox [11], Orbot [12] focus primarily on privacy of the user. However, Tor browsers that fail to consider app integrity as a security focus are vulnerable to malicious attacks. Additionally,

such apps provide little service to individuals resident in countries censuring the Tor service. Admittedly, privacy offered by Tor is remarkably sophisticated to Congo's attempt. However, Congo provides coverage for both privacy and security.

3.2.2 Multi-Process Architecture

Browsers such as Chrome for Android [13] use a multi process architecture for tabs. Tabs run as deprivileged independent processes. Compromise of a tab does not daisychain into corruption of the parent or sibling processes. However, the deprivileged process is still susceptible to privilege escalation attacks that exploit its access to Java reflection APIs, shell, and native code [14]. Consequently, isolation is important yet not comprehensive. Contrastingly, Congo is able to block reflection attacks or shell code run attempts.

3.2.3 Sandbox + App Architecture

There have been a number of papers on permission regulation and sandboxing on Android. The most robust solutions such as Apex [15], POLUS [16], CRePE [17], Saint [18] require modification of Android source to achieve unparalleled extension of the operating system. The number of distinct Android flavors to support explodes in complexity when fragmentation is considered. Over 24,000 distinct Android devices [19] were encountered in 2015 (Contrastingly, no more than 30 distinct iOS device types are currently active). Consequently, portability is hampered as accommodations not only span across different Android versions, but also consider variations within a version. Moreover, user cognitive friction is increased because installation of augmented source requires flashing - an act that exceeds most users' modest technical abilities.

Alternatives that require extensive binary rewriting, such as miAdBlocker [20] or AppGuard [21], are particularly limited in efficacy. Such techniques intimate familiarity with the target application. Additionally, malware can easily sidestep protections through dynamic loading of modules. Certain protective measures, such as PatchDroid [22], rely on function hooking enabled through ptrace. Unfortunately, such techniques have great overhead accrued from the tracing of function calls and subsequent redirection. Use of ptrace also requires root privileges.

Techniques that enforce supervision through separate apps are highly portable [23]. However, such techniques incur runtime overheads because the reference monitor is a fully fledged independent application. Additionally, latency is increased as service requests have to be relayed to and from the mediating monitor application. Consequently, a sandbox for a browser would involve running two apps concurrently: the sandboxing reference monitor and subordinate browser. Furthermore, timing attacks can exploit race conditions to hook malware before the reference monitor fully initializes. Most Android anti-virus applications [24, 25] are less sophisticated implementations of the independent invigilator. Antivirus apps pattern match against malware signatures. Consequently, certain classes of exploitation that rely on *data-execution-as-code* are undetected. In browsers, *data-execution-as-code* is a prominent attack vector employed in specially crafted JavaScript or embedded files.

4 Congo Architecture

The Sandbox comprises:

- Interposition Mechanism
- Reference Monitor (Kinshasa)

4.1 Formal Realization of Framework

1. REQUEST
   A *request*, τ, is a 2-tuple $(requester, resource)$ that associates a resource with the requester.

   A page, www.bing.com, requesting location access is denoted $(www.bing.com, android.location)$

2. ACCESS
   access is a singleton of the set $\{allow, deny\}$.

3. RULE
   A rule, R, is a tuple $(resource, access)$ that describes the availability of a *resource* as governed by *access*.

   A rule, R', denying access to the camera would be denoted as the tuple $(android.hardware.camera, deny)$.

   Contrastingly, a system with no defined rule for camera access is governed by the implicative rule $(android.hardware.camera, \emptyset)$

   A targeted rule for a specific website such as google.com would be $(android.hardware.camera, allow, google.com)$

4. OVERRIDE
   An Override, V, is a function of a rule, R, that generates a new rule R' with a different *access* singleton

$$\begin{aligned} V(R) &= V(resource, access) \\ &= R' \\ &= (resource, access') \end{aligned}$$

   maps.google.com, may request location permissions. Assuming a priori existence of a deny rule for location services, Congo will block the

request and notify the user. The user, however, may elect to allow location access thereby overriding the original $(android.location, deny)$ rule.

5. POLICY
   A policy, P, is a function that generates a subset of a set of rules. It behaves similar to a partition function with the relaxed requirement that an element may belong to multiple subsets i.e. a rule may belong to multiple policies.

   $S = \{R_1, R_2, \ldots, R_n\}$ finite set of all rules
   Z Internet Policy
   $Z(S) = S'$ set of Internet rules

6. CONTEXT
   A *context*, C, is a set of determinable environment variables.

   $$C = \{location, time, history, \ldots, network\}$$

7. REFERENCE MONITOR
   Reference Monitor, Φ, is a function of *context*, a *request*, and a *policy*. It maps the 4-tuple $(request, rules, policy, context)$ to a bit. A bit is analogous to the "yes/no" decision on a request.

   EXAMPLE 1

   $r_1 = (android.hardware.camera, deny)$
   $r_2 = (android.location, deny)$
   $S = \{r_1, r_2\}$ Rules set
   P Camera Policy
   C Context set
   $\chi = (m.facebook.com, android.hardware.camera)$

   Φ handling the request χ has the functional form

   $$\Phi(\chi, S, P, C) = \emptyset$$

   The result, $\emptyset$, dictates `m.facebook.com` is denied access. The logic can be iteratively broken down. On receiving the χ request, Φ applies Camera Policy, P, to obtain the relevant rules.

   $$P(S) = r_1 \quad (8a)$$

   Rule, r_1, specifies that the access to *android.hardware.camera* is *deny*. Consequently, the camera access request is denied.

   EXAMPLE 2

   $r_1 = (android.hardware.camera, allow, Monday)$
   $r_2 = (android.location, deny)$
   $S = \{r_1, r_2\}$
   P Camera Policy
   $C = \{Tuesday, 38°N, 77°W\}$ Context set
   $\chi = (fb.com, android.hardware.camera)$

   Φ handling the request χ will yield.

   $$\Phi(\chi, S, P, C) = \emptyset$$

   As demonstrated in *Example 1*, Φ employs the policy function to obtain relevant rules for access. This distills to the rule tuple $(android.hardware.camera, allow, Monday)$ that can be verbalized as *only allow camera access on Monday*. However, based off current context C, request χ is denied as the current day is *Tuesday*.

4.2 General Interposition Mechanism

Interposition allows a middle person to mediate whether a requested action is legal or appropriate. `Congo` intercepts system calls to *libc* - the Android standard C library. For a normal app, a Java method call that opens a file will resolve to an `fopen` in the chain of causation. For `Congo`, a Java method call will resolve to `sandbox_fopen` which is a wrapper function around `fopen`. Function hooking allows swapping out `fopen` for `sandbox_fopen` in `Congo`'s *Procedure Lookup Table*. Interception occurs at the Native code level.

Motivation for Native Code Interception: The Android stack affords Java-level or Native-level entry points for interception. Interception at Java-level is an inexpensive alluring method; however, it has some shortcomings. Interception of Java methods would require auditing and tracing of a considerable amount of methods. For instance, a desire to intercept file read/write operations would require an examination of 51 candidate methods in `Java.io` package [26]. The explosion of targets in Java-land begets an examination of a better alternative. Native-level interception becomes a tractable solution based on the observation that Java API calls resolve to C calls. Consequently, all the cited 51 methods resolve to C functions: `open`, `close`, `read`, and `write`. Another disadvantage is that Java-level supervision can easily be sidestepped through *Java reflection* or sideloading of dex files. Therefore, the standard C library of operations is a great chokepoint.

Function Hooking: Function hooking is a staple technique. It has been extensively used for means both

positive (sandboxing) [27] and nefarious (malware) [6]. Hooking itself is not particularly novel, but the means by which Congo achieves it is. Unlike other attempts at Android hooking, Congo's hooking imitates the Android kernel's dynamic linker and loader - the *Android Crazy Linker*. As a consequence, coverage is increased to all modern versions of Android (versions later than Froyo). Another side-benefit is that the hooking mechanism is genericized well enough to work on any Android app.

Hooking exploits the fact that Android object files follow the *Executable and Linkable Format* (ELF) specification. When an executable is compiled, its references to externally defined functions, i.e. `open, fork`, are initialized as stubs. These stubs are in a designated region on the binary known as the *Procedure Linkage Table* (PLT). Stubs are recorded in the PLT because the eventual memory mapped addresses of functions are indeterminable at compile time.

At runtime, the linker-loader primes the binary for execution by performing runtime binding. Specifically, PLT stubs are updated to properly map a function name, a.k.a symbol name, to the now known address of the function. A promising idea would have been to compel the linker-loader to write in bespoke addresses on custom targets. However, since the linker-loader is a very essential core of the OS, it is highly privileged and protected. Consequently, modifying the linker-loader can only be feasibly achieved through expensive means such as rooting the device or building from modified source. A viable alternative is to instead wait for linker-loader to transfer control to the binary and then overwrite the loader's entries.

Congo's hooking module is invoked immediately after the linker-loader cedes control. The module traverses through the process's virtual memory space remapping PLT entries. PLT mappings to standard C calls such as `open` and `fork` are remapped to corresponding `sandbox_open` and `sandbox_fork`. *Algorithm 2* shows the pseudocode for function hooking.

```
1:  procedure HOOK
2:      for each memory_region in virtual_memory do
3:          if SKIP(memory_region) is true then
4:              continue
5:          end if
6:          elf ← LOAD_ELF(memory_region)
7:          UNPROTECT(elf)
8:          relocation_table ← LOAD_TABLE(elf)
9:          for each symbol in relocation_table do
10:             if symbol.name in {'open','fork','execvp',...} then
11:                 symbol.address = SUBSTITUTE(symbol.name)
12:             end if
13:         end for
14:         REPROTECT(elf)
15:     end for
16: end procedure
```

2: Function Hooking

4.3 Binder Interposition Mechanism

Reliably and effectively intercepting the binder was a remarkable challenge with regard to both lack of documentation and seeming technical infeasibility.

The Android binder process intercommunication mechanism mostly espouses the microkernel philosophy. It is characterized by message passing between user applications and user space based services that relay to a microkernel. The binder framework is instantiated through *libbinder.so* (user space library) and */dev/binder* (a global readable/writeable device driver). Common app messaging mechanisms, such as *Intents* and *ContentProviders*, are built on top of binder. Congo leverages the fact that any complex communication or objects are marshalled into discernible buffers. For instance, an app requesting to send an SMS uses the binder framework to issue a request that contains a `binder_transaction_data` struct shown below.

```
struct binder_transaction_data {
 union { size_t handle; void *ptr; } target;
 void *cookie;
 unsigned int code; unsigned int flags;
 pid_t sender_pid; uid_t sender_euid;
 size_t data_size; size_t offsets_size;
 union {
    struct {
      const void *buffer;
      const void *offsets;
    } ptr;
    uint8_t buf[8];
 } data;
};
```

The key struct members are `unsigned int code` (index to function to be invoked) and `ptr.buffer` (a buffer that comprises a descriptor of requested service's interface and arguments to requested function). For the previously mentioned SMS example, the corresponding values for the highlighted struct members would be:

- `unsigned int code` is 5. An index of 5 resolves to the `sendText` method that `ISms` interface declares.
- `ptr.buffer` contains a unicode string, `com.android.internal.telephony.ISms`, which is the interface descriptor. Coming after the descriptor are the actual text message and recipient's phone number respectively.

The resolution of complex Java invocations to native code again provides an important entry point for inter-

position. Unlike techniques that require separate concomitant services or applications to parse Android's *Interface Definition Language*, the employed interposition technique allows for tight coupling of binder interception with reduced overhead. Additionally, `Congo`'s technique is portable across Android versions. Our technique intercepts at the point where data has been parceled and is about to be submitted to the binder driver. Pseudocode for binder interposition is shown in *Algorithm 3* below.

```
1: procedure INTERCEPT_IOCTL(Driver_File_Descriptor, MarshalledData)
2:     if Driver_File_Descriptor == BINDER_F_D then
3:         if MODIFY_OUTGOING(MarshalledData) == VIOLATION then
4:             return
5:         end if
6:     end if
7:     IOCTL(Driver_File_Descriptor, MarshalledData);
8:     if Driver_File_Descriptor == BINDER_F_D then
9:         MODIFY_INCOMING(MarshalledData)
10:    end if
11: end procedure
```

3: Binder Interposition

Binder interception involves modification of incoming and outgoing data. For dangerous calls, the potent parcel payload may be substituted with an empty payload that raises an invocation failure. The ability to parse the buffer affords many opportunities for identification of dangerous behavior. For instance, surveillance of Internet connections is a two-ringed defense. The first defense is binder interposition that allows for parsing of Android parcels (binder buffers). A request to connect to a hypothetical `www.android.infection.agent.com` will be identified in `ptr.buffer` and denied. Techniques such as encryption of strings may, however, mask such behavior. Fortunately, the second ring of function hooking acts as a last defense. Despite the fact that a malicious url may be masked in transit, it has to be decrypted for standard communication. Consequently, the url is unmasked when used in conjunction with C networking functions. The native C functions in this instance are `Kinshasa`'s hooked functions that screen and protect from malicious behavior. As a result, potential peril is avoided.

We later show that interposition, though seemingly invasive and expensive, adds reasonable overhead to normal operations.

4.4 Kinshasa Reference Monitor

Implementation-wise, reference monitoring is tightly coupled with interposition. *Algorithm 3* shows a surmise of interposition's integration with `Kinshasa`. The reference monitor merely provides a wrapper about the original function. In *Algorithm 3*, `modify_outgoing` evaluates the safety of a binder call. If unsafe, the potentially dangerous invocation is cancelled and does not reach the `ioctl` call. Cancellation is not the only possible response to dangerous requests. A phonebook request may be allowed to pass only for the consequent reply to the request to be modified. In this instance, the legitimate phone book in the reply is substituted with a dummy phone book. Despite deep coupling of code, the breadth of supervision can be better understood through `Kinshasa`'s policy framework.

4.4.1 Policy Framework

Privacy: This policy has the dual intent of undergirding user and application data confidentiality.

- *User Privacy* - Binder requests to confidential service providers, such as the phonebook, are subject to the reference monitor. The reference monitor gives the user the option of granting or denying access. However, not all requests are mediated through the user. Certain contexts have enough information for user-excluded mediation. On strict permissions, if a device on public wifi receives a request for location from an unknown website, the request is denied without user involvement.
- *Application Data* - Interposition of I/O tools i.e: `open`, `read`, `write` allows for mediation of filesystem reads and writes.

Network Communications: This policy monitors the following external communication channels:

- *Internet* - Connection attempts to websites are mediated. A blacklist of known malicious or adware-laden targets are used to referee Internet connection requests.
- *Cellular* - Interposition of binder requests to telephony services aims to prevent abuse of SMS and phoning features. Malicious ventures use SMS and phoning as covert communication channels or as means to rack up charges on premium phone services.

Execution: This policy prevents remote code execution. The following channels for privilege escalation are blocked:

- *Reflection* - A malicious actor may use reflection to gain access to classes and methods whose visibility is hidden through access modifiers i.e. private or protected.
- *Shell Runtime* - The Android runtime ordinarily allows access to bash runtime. Such facility is particularly dangerous for rooted users who normally have root user enabled with the default password. At first glance, non-rooted devices may seem immune to shell chicanery. However, a clever attacker can invoke or chain standard shell commands, i.e.

`ls`, `grep`, to induce clinical stack and buffer overflows for privilege escalation. Additionally, seemingly innocuous commands, such as `ls`, can be used to fingerprint a device.
- *Native Code* - C library functions that are standard ingredients for privilege escalation techniques, i.e. `dlopen`, are made unavailable to `Congo`. `Kinshasa` responds with failure to invocation requests for such functions.

Context: This policy utilizes a context manager to enable proactive mitigation of threats. The context manager works across domains such as:
- *Frequency* - URLs not in browsing history are scrutinized greatly. Such unusual URLs are only loaded after an advance warning is displayed and the user elects to proceed with visiting the URL.
- *Location* - Locations are classified similar to the Windows OS designation of private and public network locations. Public unsecured networks are treated as insecure by default and requests for privilege and privileged data are not honored.
- *Time* - Certain privileges can be availed at only certain times as demonstrated in *Example 2*. Additionally, rules can be created with expiration dates. For instance, a user could disable location services for an hour. Consequently, *Kinshasa*, within the hour window, will reject location requests unless explicitly overruled by the user.

5 Evaluation

Efficiency of a proposed mobile framework is critically important; especially because mobile devices are resource constrained. As a result, evaluation is performed through a series of repeated trials on different metrics. `Congo` is tested for Runtime Robustness, Performance Impact, and Sandbox Enforcement. Tests reported are performed on a Nexus 5 on Android 4.4, the most popular version of Android on active handsets. Additionally, with a view to corroborate results, evaluations are also performed on a Nexus 6 running Android 5 and emulators running the modern version families of Android; namely, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat, and Lollipop.

5.1 Runtime Robustness

To asses robustness, `Congo` is automated to load the top 500 [28] websites globally. Each website is subject to 50 input events to mimic navigation about the website. 4.2% of websites crash upon launch or navigation. 64% of the crashes are attributable to general website noncompliance with the webkit rendering engine. The remainder, 36%, of the crashes are because of websites requesting unanticipated permissions which were ungracefully denied. However, a success rate of 95.8% for a dynamic list of the most popular websites globally is a promising achievement for `Congo`. The success rate can further be improved through a complicated robust overloading of the JavaScript V8 engine or a simple ECMAScript interpreter.

5.2 Performance Impact

Table 1 and Table 2 show the results of benchmarking Android API calls and syscalls.

sycall	Native(μs)	`Congo`(μs)	Overhead(%) $= \frac{Congo-Native}{Native}$
create	34.5	40.3	16.81
open	6.4	12.1	89.06
ls	5.9	11.2	89.83
rm	95.6	100.7	5.33

Table 1: Timed Syscall Microbenchmarks

For syscalls, it is observed that the overhead is an almost constant 5μs which represents the average duration for a local target request to be handled.

API	Native(ms)	`Congo`(ms)	Overhead(%) $= \frac{Congo-Native}{Native}$
Create Socket	119.8	130.9	9.27
Obtain Location	40.8	51.56	14.66
Query Contacts	10192.2	10051.9	1.38

Table 2: Timed API Microbenchmarks

Unsurprisingly, API calls incur the most overhead. The overhead is reasonably under 15% and quite similar to Boxify [29]. The reason for the substantial overhead compared to syscalls is that API calls require interception of IPC packets (Android parcels). Packet interception is expensive in memory and time since it involves cloning of packets and an accompanying shuttling from native-level to Java-level and back. The demonstrated overheads, however, are the worst case scenario. Ordinarily, not many remote IPC-based APIs are used by Android applications; especially `Congo`. As a result, the 15% is not a constant recurrent bottleneck. Additionally, once an API handle is acquired, future invocations of the API are not invigilated for the handle's duration.

5.3 Sandbox Enforcement

To test actual protection from malware and malvertising, `Congo` is tested against 20 variant techniques used to exploit user privacy and security on Android [6]. `Congo` is able to successfully defend against security and privacy

leaks. Due to time limitations and difficulty in tracking fully fledged malicious websites, the sample set was kept at 20. Despite the seemingly small sample size, the set was found to be quite representative of the commonest malicious methods.

The following JavaScript listing is a malicious sample that Congo successfully protects against.

```
1  //https://www.exploit-db.com/exploits/
       32884/
2  function execute(bridge, cmd) {
3     return bridge.getClass().forName('java.
          lang.Runtime')
4       .getMethod('getRuntime',null).invoke(
            null,null).exec(cmd);
5  }
6  if(window._app) {
7     try {
8       var path = '~/gotcha.txt';
9       execute(window._app, ['/system/bin/sh
            ','-c','echo \"MITM Done\" > '
10 + path]);
11      window._app.alert(path + ' created',
            3);
12    } catch(e) {
13      window._app.alert(e, 0);
14    }
15 }
```

Listing 1: Sample of malicious Javascript

An unprotected web browser rendering a page containing *Listing 1* invites an innocuous but unsanctioned filesystem write. The demonstration can easily be scaled to more malicious purposes. A particularly potent threat is chaining bash commands to induce buffer overflows in POSIX-defined binaries.

Congo, however, denies the script privileges and thus maintains device integrity. Particularly, Kinshasa intercepts a request to the `java.lang.Runtime` class and returns an exception to the caller. The `java.lang.Runtime` rule is hardcoded to always deny because Congo should never have the need arise for direct invocation of shell. As a result, the exception thrown by Kinshasa is passed back up to the callee function, `execute`, resulting in a Javascript exception instead of the originally targeted filesystem write.

6 Future work

Future effort can be expended into overcoming current shortcomings:

- *Augmented Context Manager* - Machine learning can be incorporated to train on a user's behavior and develop a more sophisticated model that allows for better recognition of atypical, and possibly malicious, behavior.
- *Improved Integrity Protection* - Malware resident on a device can exploit timing to hook and inject its own code before Congo hooks. Malware residence on the device is beyond the scope of Congo, however, a canary can be instrumented to assist in detection of preload tampering.
- *Sandbox Instrumentation Framework* - The coupling between vanilla Java and native code comprising the sandbox is minimal. Consequently, the sandbox can be made generic enough such that it can be weaved into any other Android binaries using a binary instrumentation framework.

7 Conclusion

Congo is the first dedicated hardened browser on Android. Congo implements a context-aware sandbox that contains malicious activity proactively and mitigatively. The virtual sandbox is achieved through through use of interposition of functions and Android IPC communication system . In addition, Congo is particularly portable. Unlike extensive sandbox implementations that modify source or need root access, Congo is implemented as a shared library that is greatly portable and works on versions of Android later than 2.1. A great consequence of the non-invasive implementation is that user friction is eliminated as users need not root or flash their phones to make utility of Congo. Congo demonstrably runs as a normal Android application with minimal overhead on resources. Additionally, experimental testing shows remarkable success in protecting against various attempts at privilege escalation or user fingerprinting.

References

[1] IDC Research, "Smartphone os market share," 2016. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] Ernst & Young Global Limited, "2015 global information security survey," 2015. http://www.ey.com/Publication/vwLUAssets/ey-global-information-security-survey-2015/$FILE/ey-global-information-security-survey-2015.pdf.

[3] R. Nigam, "A timeline of mobile botnets," 2015. https://www.virusbulletin.com/virusbulletin/2015/03/timeline-mobile-botnets.

[4] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning, "Andbot: Towards advanced mobile botnets," in *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, LEET'11, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2011.

[5] E. S. Raymond, "The cathedral and the bazaar," 1997. http://www.unterstein.net/su/docs/CathBaz.pdf.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th International Conference on Information Security*, ISC'10, (Berlin, Heidelberg), pp. 346–360, Springer-Verlag, 2011.

[7] T. Hill, B. Schmertz, C. Hughes, and R. O'Dea, "Advanced risc machines," 2008. https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm.

[8] The Chromium Authors,The Android Open Source Project, "A custom dynamic linker for android programs," 2015. https://chromium.googlesource.com/android_tools/+/master/ndk/sources/android/crazy_linker.

[9] Google Inc, "Welcome to the android open source project," 2016. https://source.android.com/.

[10] Google Inc, "Android for work security white paper," 2015. https://static.googleusercontent.com/media/www.google.com/en/US/work/android/files/android-for-work-security-white-paper.pdf.

[11] The Tor Project, "Orfox:tor browser for android," 2015. https://play.google.com/store/apps/details?id=info.guardianproject.orfox.

[12] The Tor Project, "Orbot:proxy with tor," 2016. https://play.google.com/store/apps/details?id=org.torproject.android&hl=en.

[13] Google inc, "Google chrome for android," 2012. https://developer.chrome.com/multidevice/android/overview.

[14] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[15] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints."

[16] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 271–281, IEEE Computer Society, 2007.

[17] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: Context-related policy enforcement for android," in *Proceedings of the 13th International Conference on Information Security*, ISC'10, (Berlin, Heidelberg), pp. 331–345, Springer-Verlag, 2011.

[18] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, (Washington, DC, USA), pp. 340–349, IEEE Computer Society, 2009.

[19] OpenSignal, "Android fragmentation visualized," 2015. http://opensignal.com/reports/2015/08/android-fragmentation.

[20] K. El-Harake, Y. Falcone, W. Jerad, M. Langet, and M. Mamlouk, "Blocking advertisements on android devices using monitoring techniques," in *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pp. 239–253, 2014.

[21] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard: Enforcing user requirements on android apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, (Berlin, Heidelberg), pp. 543–548, Springer-Verlag, 2013.

[22] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, (New York, NY, USA), pp. 259–268, ACM, 2013.

[23] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: Fine-grained permissions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, (New York, NY, USA), pp. 3–14, ACM, 2012.

[24] Avast Software, "Mobile security & antivirus," 2016. https://play.google.com/store/apps/details?id=com.avast.android.mobilesecurity&hl=en.

[25] AVG Mobile, "Avg antivirus free for android," 2016. https://play.google.com/store/apps/details?id=com.antivirus&hl=en.

[26] Oracle and/or its affiliates, "Package java.io," 2015. https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html.

[27] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 539–552, USENIX, 2012.

[28] Alexa Internet Inc, "The top 500 sites on the web," 2016. http://www.alexa.com/topsites.

[29] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 691–706, USENIX Association, Aug. 2015.