Variations in WebView (Public Version)

Author: paulmiller@
Reviewed by: -
Last Updated: 2018-05-18

One-page overview

Summary

We're adding Variations (Chrome's experiment system) to Android WebView (the Chromium-based Android widget for embedding web content in Android apps).

Platforms

Android

Team

paulmiller@ (eng), sbirch@ (PM)

Launch bug

https://crbug.com/793888

Code affected

WebView (src/android_webview/) and Variations (src/components/variations/).

Motivation

PlzNavigate exposed WebView's lack of Variations as a major risk to stability and drain on engineering time. While PlzNavigate was ramped up experimentally in Chrome, we had to launch from 0% to 100% in WebView. This led to many stable-channel bugs due to incompatible behavior changes. While PlzNavigate was especially bad, every WebView feature (and many Chrome features which affect WebView) carries this same risk.

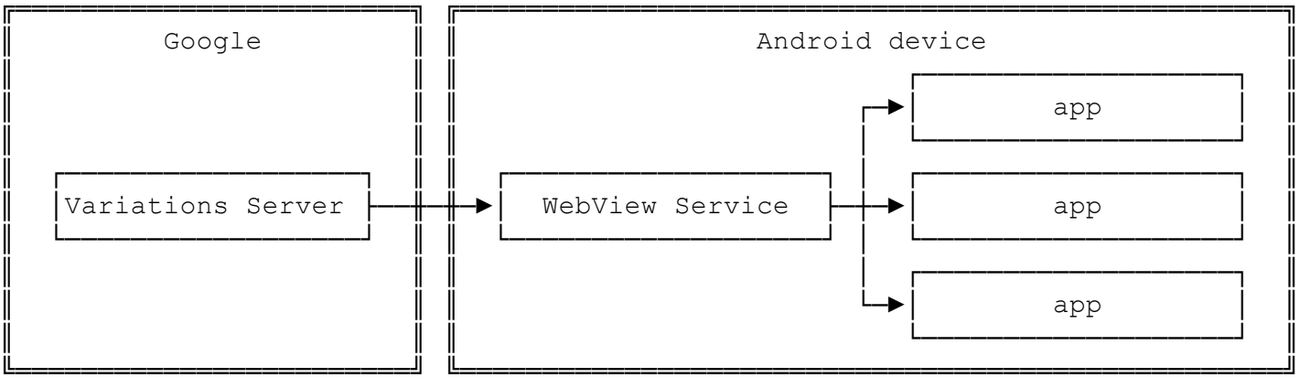
Design

Experiment setup

The process for creating experiments, and the server-side Variations implementation, will be the same across Chrome and WebView. "Android WebView" has been added as a separate platform, alongside "Android", "Windows", etc. This implies that existing experiments (which were not explicitly configured to include WebView) will not automatically apply to WebView. It also implies that WebView's Variations seed (the list of experiments and the circumstances for enabling them) will differ from Chrome's.

Getting the seed

Like Chrome, WebView must periodically download a Variations seed from Google. Unlike Chrome, WebView runs inside the processes of many other Android apps; WebView must disseminate the seed to all WebView-using apps on the device. We handle this in a new service, running in a separate process, which downloads seeds and disseminates them to apps on request:



The Variations Server here is the same server that Chrome uses.

Neither the Server→Service nor Service→App transfer is guaranteed to complete successfully, and may result in a truncated seed. Therefore the service and every app will each keep 2 seed files: "old" and "new". New seeds will be written to the "new" file, and later the "old" file will be replaced with the "new" file. If the "new" seed is found to be truncated, it will be ignored and the "old" seed will remain in use.

The seed file will be a serialized proto containing the same fields as VariationsSeedFetcher$SeedInfo: the seed data, the signature and compression flag for that data, the country code, and the date the seed was downloaded.

The proto wire format is such that if the file is truncated in the middle of a field, parsing will report an error. If the file is truncated between fields, no error will be detected. However, we expect to see every field in the seed file; therefore if not all fields parse successfully, we will know the seed was truncated.

Loading the seed in the app

Variations' design is such that the experiments must be configured on startup and can't change while Chrome/WebView is running. Therefore we need to block WebView startup on loading an app's local copy of the seed. Loading the seed takes < 2 milliseconds on Nexus devices, but this may be worse on low-end devices. There will be a 20 millisecond timeout; if the seed is not available in time, WebView will continue with all Variations experiments disabled. (Total WebView startup times varies greatly across devices, but I've measured it to be 100-200 milliseconds on Nexus devices.)

Because of this timeout, and because no app will have a seed the first time it uses WebView, we can't guarantee seed availability. Therefore, enabling a feature at 100% doesn't guarantee it will be enabled for everyone. Similarly, any feature that defaults to "on" cannot reliably use Variations as a killswitch.

Requesting a seed from the service

If the app doesn't have a seed, or its seed is old, the app will request a new seed from the Service. We can't block startup on this; if the Service process isn't already running, starting it takes hundreds of milliseconds.

Apps will request the seed via this Binder call:

```
interface ISeedServer {
    void getSeed(in ParcelFileDescriptor dest, long date);
}
```

The app will supply an open file descriptor for its "new" seed file, to which the Service will write the new seed. The app will report the "date" field of its current seed, if any; the Service will only write a new seed if it's newer than what the app already has. The app will rate-limit its requests across runs by writing the time of its last request in its data directory.

Downloading a seed from Google

Whenever the Service gets a request, the Service may schedule a JobService to download a new seed. The JobService will be scheduled iff the Service's copy of the seed is sufficiently old, and the JobService is not already scheduled. The Service and the JobService will run in the same process.

The advantages of using a JobService are:

- The JobService can be made to wait for network access before running.
- The system schedules JobServices from multiple apps together to minimize battery impact.
- Since the Service is foreground, it may be killed at any time due to memory pressure. The download may take longer than the lifetime of the Service. A JobService is more likely to complete successfully.

The JobService will use Chrome's "firstrun" seed download code, implemented in Java, rather than Chrome's native downloader. This is because the WebView Service doesn't load native code, to keep it light weight. The Java downloader doesn't support delta-compression of seeds, so we'll want to add this as a future optimization. (See "Follow-up work", below.)

Rejected Designs ideas for disseminating the seed to apps

I investigated nine alternatives, falling into these categories:

Rather than loading the seed from flash, transfer it from some other process. This could be a Service or ContentProvider, and could share the seed via Binder, Messenger, or ashmem.

Unfortunately, WebView can't rely on starting another process to get the data; starting a process is slow, and we can't afford to delay WebView that much. I tested a bare-bones ContentProvider on Nexus 6, N2G47S, and found latency to range from 1-10 milliseconds when the ContentProvider's process was already running, but 150-400 milliseconds when it wasn't. I also tested binding (measured from right before the bindService call to the onServiceConnected callback); I found the latency to range from 10-20 milliseconds if the Service was already running, and 150-200 milliseconds when it wasn't.

The process could be made persistent, but that would make the process's memory cost persistent. Testing on Nexus 6 (arm32), N2G47S, WebView's variations_service process has a PSS of ~4 MB when no other apps are using WebView. On Nexus 9 (arm64), it's ~11 MB [1]. We could mitigate the impact by disabling Variations on very low-memory devices, although that would skew results for performance-focused experiments.

GMS contains a persistent process for frequently-used services, so we could serve the seed via GMS. Unfortunately, there is no existing GMS API that would support our needs. We would have to implement and maintain a new API. This would be a large and ongoing burden on the WebView team, given GMS' extensive bureaucratic and technical launch requirements. (Even simple changes require a unique, 20-strong dogfood population; myriad forms, docs, and tickets to be filed; participation in the regular integration days; etc.)

Rather than copying the seed into each app, store one copy in some globally-readable location. This could be a world-readable directory within WebView's data directory, or in /sdcard.

Unfortunately (for us, but fortunately for security), Android is moving away from globally-readable files, and pushing apps to share data via IPC mechanisms instead. MODE_WORLD_READABLE was deprecated in JB and effectively removed (it throws a SecurityException) in N. File.setReadable()/setExecutable() may be completely blocked via SELinux in a future release.

Every parent directory of the world-readable file (meaning WebView's data directory, or Chrome's in the case of Monochrome) would additionally need to permit execute permission, which weakens the app sandbox.

We can't use /sdcard because 59.06% of apps which use WebView don't declare READ_EXTERNAL_STORAGE permission [2] and so can't access /sdcard. Android is trying to move away from /sdcard as well, so this percentage is likely to decrease. Additionally, /sdcard behavior varies across devices; some have multiple external storage directories, or removable sdcards, and /sdcard's performance characteristics may differ from the rest of the filesystem.

/data/local/tmp was also recently locked down.

As above, but with evil hardlink skullduggery. The seed would live in WebView's data directory, and that directory would not have execute permission. Instead, each app would have a hardlink to the seed file in its own directory. The seed itself would be world-readable but not world-writable. (All hardlinks to a file share the same permissions.) This would be a smaller violation of the app sandbox. This (awful) hack wouldn't work with symlinks, because those traverse the original parent directories, and would be blocked by the directory's lack of execute permission.

Unfortunately, we have no process with privileges to create this link. A WebView Service wouldn't be able to access the app's directory, and the app wouldn't be able to make a link into WebView's directory. Apps can grant each other access to private files by passing file descriptors, but a link cannot be created from a file descriptor; `Files.createLink` (Java) and unistd.h's `link` (native) both take a pair of paths.

Get help from Android framework. Android could give guarantee us a globally-readable filesystem location to share the seed. Or it could arrange for the seed file to to already be loaded when app processes are forked.

However, given the uptake rate of new Android releases, we don't want to restrict Variations to future Android versions which have the necessary changes.

Rejected alternative: Phenotype

GMS has the Phenotype API, an experiment framework used by many 1st-party Android apps. We could use Phenotype rather than Variations to run experiments in WebView. But this has several problems, in order of increasing estimated severity:

- Phenotype is currently restricted to 1st-party apps. Although WebView is 1st-party code, it runs inside 3rd-party apps, and so cannot access 1st-party GMS APIs.
- Phenotype has different semantics than Variations, e.g. Variations has a 1:1 relationship between flags and experiments, whereas Phenotype does not. This would complicate experiment design.
- This would diverge WebView from Chrome, complicating launches which cover both platforms. The differences may alienate non-WebView members of the Chrome team, and discourage proper experimentation.

We plan to revisit this option if our experimental results (see "Experiments" below) are unmitigably poor.

Metrics

Success metrics

We'll add WebView logs for the existing Variations.SeedFreshness histogram to verify prompt seed delivery to apps.

Regression metrics

WebView startup is not currently instrumented; we will add UMA histograms to measure time spent blocked on loading the seed, and success rate of loading the seed (how often we time out). AGSA (See "Experiments" below) has its own performance metrics which they will monitor.

Experiments

This is a catch-22: we can't run experiments in WebView until this feature is complete. So we've instead created an experiment in AGSA (Android Google Search App). AGSA uses WebView to show search results and his highly sensitive to WebView startup latency. AGSA instances inside the experimental group will

touch a file in their data directory to indicate that WebView should enable Variations. AGSA will monitor their own metrics to assess WebView's impact on the host app's performance.

We aim to experiment in m67. Because of WebView's small beta population, we'll have to wait for the experiment to reach stable to get meaningful data.

The entire implementation need not be complete to begin experimentation; loading the seed from flash on startup, and starting the WebView Service process shortly after startup, are the steps expected to impact WebView performance. Downloading actual seeds and running actual Variations studies are not necessary at this stage.

Rollout plan

See "Experiments", above. If the AGSA experiment results are acceptable, we'll enable Variations in WebView for all apps, not just AGSA.

Core principle considerations

Speed

These factors may impact performance:

- WebView startup time may be delayed by loading the seed.
- WebView's services process is currently only run when uploading crashes; with Variations, this process will run more frequently (every time an app requests a new seed), and this work will likely share CPU time with the host app's startup (since many apps create WebViews on startup).
- The JobService to download new seeds will use network up to once per day.

Startup impact is mitigated by the 20 millisecond timeout. Impact on the host app will be measured in the AGSA experiment.

We plan to delta-compress seed downloads as a future optimization (see "Follow-up work", below) but are not including this in the initial launch. WebView's current seed size ~3KB, much smaller than Chrome's ~260KB. This optimization will become more impactful as WebView's seed grows from new experiments.

Security

The seed is downloaded via HTTPS and stored in WebView's private data directory. Android's app sandboxing prevents other apps from reading or writing to this directory. The `getSeed` Binder API described under "Design" may be called by any installed app. Therefore any app can observe but not modify the WebView Service's copy of the seed. If some party were able to modify the seed, they would be able to enable/disable whatever experiments were implemented in WebView at the time, for all WebViews on the system. If there are any bugs in the protobuf library we use to parse the seed, they might be able to exploit those with a specially-crafted seed.

Any app can modify its own copy of the seed, since it's stored in the app's own data directory. So it's possible for apps to enable/disable experiments for their own WebView instances, without affecting any other apps on the system. However, since WebView runs inside the processes of other apps, apps already have complete control over their own WebView instances.

(Bonus 5th S) Space

The main drawback of this design is the cost of copying the seed into every app that uses WebView, on flash space and write endurance.

74.45% of apps use WebView [2]. If a user uses 100 unique apps, and we naively assume 74 of those apps use WebView, then upon pushing a new 3 KB seed, their device would write 222 KB to these apps. The writes would occur intermittently as each app was used. If over years of adding experiments, WebView's seed were to eventually grow to match Chrome's 260 KB, the cost would rise to 19 MB, which may be unacceptable.

Chrome is working to reduce seed size by perhaps 80%, by removing expired experiments, and filtering out experiments by target milestone and channel. (See https://crbug.com/761684.) WebView can share in these optimizations. Still, we'll need to monitor the growth of WebView's seed over time.

Privacy considerations

WebView already collects UMA histograms based on Android's metrics consent setting (the checkbox shown in Android's setup wizard). Just like in Chrome, Variations will annotate WebView's UMA data with which experiments were enabled at the time.

Since the set of enabled experiments is decided randomly by the client, a Googler with raw logs access could use experiment info to fingerprint a given (app, user, device) tuple. However, UMA already uploads a "client ID", a randomly-generated number for pseudonymously identifying UMA data (unique per (app, user, device) tuple). The client ID contains 122 random bits, sufficient to avoid collisions; it is an accurate fingerprint by design. An experiment-based fingerprint would have a number of random bits equal to:

$$\log_2(\prod_{i=1..n} G_i)$$

(where n is the number of experiments, and G_i is the number of assignable groups in experiment i). This would give it equal or lesser accuracy compared to the client ID, which already has total accuracy.

Testing plan

Variations is not visible to the user, so testers need not verify any user-visible behavior. Instead, testers should enable Variations in WebView via flag, to exercise Variations during their normal testing and check for crashes:

```
$ adb shell 'echo "_ --enable-webview-variations" > /data/local/tmp/webview-command-line'
```

Testers should also verify that the AGSA experiment functions with the latest WebView versions. Testers should verify that when the experiment is enabled, the "finch-exp" file appears, and that WebView creates a dummy seed file when the "finch-exp" file is present.

The specific steps would be:

1. Get access filesystem access:
   ```
   $ adb root
   $ adb remount
   ```
2. Clear any existing WebView seed files from AGSA:
   ```
   $ adb shell rm
   /data/data/com.google.android.googlequicksearchbox/app_webview/variations\*
   ```
3. Install the latest "release" build of of AGSA and NowDevUtils.
4. Restart AGSA (force stop in settings and then do a search).
5. Verify that the "finch-exp" file doesn't exist:
   ```
   $ adb shell ls /data/data/com.google.android.googlequicksearchbox/files/webview
   ```
6. In NowDevUtils, side menu "Flags", select "Manage config flags" and search for AgsaWebview__finch_experiment_file_policy.
7. Change the value of the finch_experiment_file_policy to 1 (create file)
8. Restart AGSA.
9. Verify that the "finch-exp" file exists.
10. Wait a few seconds for WebView to create the seed file.
11. Verify the file was created:
    ```
    $ adb shell ls /data/data/com.google.android.googlequicksearchbox/app_webview/
    ```
    There should be a "variations_seed_new" file there. (Further usage may cause the file to be renamed to "variations_seed".)
12. In NowDevUtils change the value of the finch flag to 2 (remove file)
13. Restart AGSA.
14. Verify that "finch-exp" no longer exists.

Follow-up work

There is code to enable Variations based on a command-line flag or the AGSA experiment which will be removed if the current implementation proves acceptable.

Once we've gotten experience running an actual experiment for WebView, we'll add documentation about the process and any differences from Chrome.

There are additional features and optimizations we hope to add in the future:

- Support persistent filtering of experiments by country: https://crbug.com/823410
- Delta-compress Variations seed downloads: https://crbug.com/817506
- Support falling back to an old seed if a new seed causes problems: https://crbug.com/801771
- Include experiment info in WebView crash data: https://crbug.com/758450

Notes

[1] Memory usage by WebView's variations_service process was measured like so. To start the Service:

```
$ adb shell am startservice
com.google.android.webview/org.chromium.android_webview.variations.AwVariationsConfigurationService
```

To measure memory:

```
$ adb shell dumpsys meminfo com.google.android.webview:variations_service
```

Quoted numbers are PSS, while no other processes are using WebView (which is the worst-case scenario).

[2] Stats on how many apps use WebView, and how many declare which permissions, were found via Play Store data.