APPENDIX A
ARTIFACT APPENDIX

WebViews are a common mechanism for embedding web content within Android applications. Although they provide browser-like capabilities and execute in an isolated context, apps can still directly interact with them by dynamically injecting JavaScript code at runtime. While prior work has extensively analyzed apps' Java code, existing frameworks have limited visibility of the JavaScript code being executed inside WebViews. Consequently, there is limited understanding of the behaviors and characteristics of the scripts executed within WebViews, and whether privacy violations occur.

To address this gap, in our NDSS submission, we proposed WebViewTracer, a framework designed to dynamically analyze the execution of JavaScript code within WebViews at runtime. Our system combines within-WebView JavaScript execution traces (obtained using browser instrumentation) with Java method-call information, to also capture the information exchange occurring between Java SDKs and web scripts. We used this system to crawl a dataset of 10K apps downloaded from the Google Play App Store.

Due to the highly dynamic nature of the data being observed and the scale (over a month of crawling), the experiment conducted in the paper is hard to reproduce. As a result, as an artifact we are providing a set of 195 apps, from our dataset, for which we downloaded the x86_64 version (to work with our emulated environment) that we believe will provide a good-enough approximation of the high-level results we obtained from the crawl. We also provide the original dataset of JavaScript execution traces and logs dumped by Frida's tracing of Java SDKs that we used to obtain the results in the paper.[1]

A. Description & Requirements

Our artifact consists of three parts, the first is the modified VisibleV8 patchsets that were used to create the VisibleV8 WebView. We provide detailed instructions to build and install it on a Android phone alongside prebuilt versions of the apk for x86_64 and ARM64. The second is the UIHarvester based crawler that was used to run our experiments. This uses docker to start a prebuilt emulator that already has a version of the VisibleV8Webview installed. This component allows us to exercise the apps and helps obtain the Frida logs used to perform our experiment. The third component of the artifact is our dataset. We provide a `apk_artifact.json` file that contains the hashes, version and app names of all the apps used in our experiment as well as a small zipped subset of 195 apps that can be used to run a small scale experiment.

To build our dataset of 195 apps, we took the list of apps that had webviews from our original dataset, picked 500 apps at random, and then re-downloaded the apps from the Google Play App Store using the credentials of the emulated phone. Out of these 500, 198 apps were successfully downloaded for the specific architecture and credential combination on July 14th 2025.

1) How to access: The artifact files can be downloaded from https://doi.org/10.5281/zenodo.16028902. The `WebViewTracer-main.zip` archive in Zenodo represents a Git repo that we will open-source with the paper. The `SystemWebView.apk` file is the prebuilt x86_64 version of the VisibleV8 WebView provider. The `trace-apis.patch` and `chrome-sandbox.patch` are the patches that can be used to build a version of VisibleV8 Chromium for v138. The APK dataset is at `x86_64_apps.zip`. The `avd.zip` is the archive containing a `x86_64` emulator that can be used to run the small scale experiment.

2) Hardware dependencies: The scaled down version of our system requires a x86_64 system with CPU virtualization support. In addition, the system running the experiment must have CPU and memory requirements to run Android Studio. [2] To replicate the larger version of the full experiment run will require at least one Android Pixel phone based on the ARM64 architecture. To run the results scripts to reproduce the experiment on the original data, a significant amount of SSD space is required.

3) Software dependencies: Docker, docker compose (higher than v2), Python $\geq$ 3.10 and venv need to be installed on the system and are necessary to run these experiments. Note that rootless docker alternatives like `podman` are not supported since they interfere with experiment setup. Python is required to run the orchestration scripts and docker is used to create databases, and setup a reproducible emulator (we tested emulator version 36.1.9.0, build_id 13823996). The emulator-based mode requires a Linux system with kernel-based KVM support as a precondition to the Android emulator working. For the larger experiment, an Android Pixel phone (our experimental system used a Pixel 4a) is required that would need to be rooted with some kind of rooting software like Magisk or KernelSU and a version of "systemizer" a method of manually replacing system apps with different variants should be installed. We've tested our scaled-down experimental system on Ubuntu 24.04.01, with a docker version of 28.3.3 (build 980b856) and a docker compose version of v2.39.1. Our original large-scale experiment was also run Ubuntu 24.04.1, with a docker version of 28.2.2 (build e6534b4) and docker compose version of v2.36.2. We have even tested our system on Arch Linux (running a kernel version of 6.15.9) with a docker version of 28.3.3, build 980b856 and docker compose version of 2.39.2 (note that any kind of aliasing of docker to alternatives are not supported).

4) Benchmarks: While there are no explicit benchmarks being performed, we do provide two datasets as part of our artifact, the first one is the dataset of apps, this is a list of all apps that were downloaded and used to perform our analysis. These can be used to perform our experiment. Another dataset contains a set of files containing the browser execution traces

[1] https://doi.org/10.5061/dryad.05qfttffz

[2] https://developer.android.com/studio/install#linux

and Frida logs that were subsequently used to obtain our results.

We provide two scripts, `android-check.sh` and `python ./scripts/wvt-cli.py` (see Section B) to check if your system is capable of running the artifact. If the `android-check.sh` is capable of running and displaying an emulator window and the `python3 ./scripts/wvt-cli.py` script exits without a 255 exit code, you should be able to run the small-scale experiment.

B. Artifact Installation & Configuration

To run the scaled down experiment the following steps must be followed,

- Download the `WebViewTracer-main.zip` from https://doi.org/10.5281/zenodo.16028902
- Run `git init && git add . && git commit -m Init` inside the directory of the zip file.
- Change directory to the webviewtracer-crawler directory
- Setup a python virtual environment by running `python3 -m venv env && source env/bin/activate`
- Install the dependencies using `pip install -r scripts/requirements.txt`
- Run `python3 ./scripts/wvt-cli.py` and make sure it does not exit with a 255 exit code and does not output any error messages.
- Run the `android-check.sh` to check if you are able to correctly run a small scale emulator on your system.
- Download the AVD emulator image avd.zip in 10.5281/zenodo.16028902 and unzip it into the `celery_workers/avd/` directory
- Run `python3 ./scripts/wvt-cli.py setup`, the CLI will ask you a few questions, you can choose the default option by pressing enter once the questions are asked.
- Navigate to the "`apps/split_1`" directory and unpack x86_64_apps.zip in 10.5281/zenodo.16028902 into the directory
- Run `python3 ./scripts/wvt-cli.py crawl`. If this command does not work, please use `docker compose --env-file .env up --build -d -V --force-recreate --remove-orphans`, the python script is a transparent wrapper to this command and it should start running the experiment.
- Navigate to `http://0.0.0.0:6901` and observe the apps being crawled
- Once all the apps are done crawling, run `ls raw_logs/ > tmp` in the webviewtracer-crawler directory.
- Run `python3 ./scripts/wvt-cli.py postprocess -f tmp -pp 'Mfeatures+androidflow+exfil+frida'`, open `0.0.0.0:5559` and wait for all the tasks to succeed or fail. The jobs will fail if no webviews are loaded at all while crawling the app and it is normal for a lot of them to fail.
- Run `python3 ./scripts/wvt-cli.py results` to view the results
- You can use `python3 ./scripts/wvt-cli.py shutdown` to shutdown the crawler

Region specificity. Many of the apps that we provide have differing behavior based on the region in which they are run, (for example, when the apps are run in the Europe, Google Ads will display a consent banner to align with GDPR laws), to reproduce our experiments, we do recommend using a VPN to connect to a server in the US to more closely replicate the results we got in our paper. We include `openvpn` as part of our Docker container, you can drop a valid .ovpn file in the `vpn/` directory and then load the VPN by using `sudo openvpn /app/vpn/<config.ovpn>` inside the docker container.

Debugging steps Depending on hardware configurations and regional differences, there might be issues with emulators crashing when opening and running specific apps, we recommend analyzing the logs in the "webviewtracer-crawler/raw_logs" directory (of the unzipped WebViewTracer-main.zip file) to understand the errors that being hit during crawling an app. Each app will have its own subdirectory which will contain the a directory called "logcat/" which contain a logfile (called "full_logcat") which contains the logcat output during the run of the app. Similarly, often the issue can be incompatibilities with Frida hooking, our "raw_logs" dumps also contain the logs of all frida calls in "frida/logfile". Finally, every logfile contains a set of images taken during the traversal of the app itself, which can verify how the crawling setup navigated the app and a set of raw VisibleV8 logs in the "Documents" directory for each app which can be inspected to understand issues with JS execution inside the WebViews. These logs later get postprocessed and are used during the analysis.

Running the full-scale experiment. If you would like to run the full scale experiment on newer browser versions, you can find newer version of our VisibleV8 WebView patches in the upstream VisibleV8 repo[3]starting with Chrome 139. The patch files provided in the "patches" directory can also be built from scratch by following `patches/webview_build_instructions.md` of the unzipped WebViewTracer-main.zip file).

We would recommend starting with downloading a set of apps using the methodology we outline in `dataset/Downloading_apps.md` (of the unzipped WebViewTracer-main.zip file) and then setting up a ARM64 phone with VisibleV8 WebViews, before running the same commands that were run for the small-scale experiment, except that during the setup command (step 5) the 'physical' prompt must be chosen when the script asks the question "What type of devices are you using?". Note that the devices must be plugged in and connected and authorized to connect be remotely debugged through adb by the computer running the experiment.

[3]https://github.com/wspr-ncsu/visiblev8

C. Experiment Workflow

The experiment starts by setting up a set of dockerfiles and then subsequently starting an already prepared Pixel 6a emulator with WebView version 138 installed, installing apps one by one on the emulator and using UIHarvester to perform a depth-first-search of the UI of the apps. It does this for 5 minutes per app and retries them two times if there is unforeseen failure of the Frida-based instrumentation at any moment.

In the experiment conducted in our paper, we used 4 real Google Pixel 4a devices that had a the VisibleV8 Webview provider for version 131. We also used a timeout of 20 minutes and retried every single app 5 times across 1K apps. The amount of time the system spends crawling an app can be tweaked by editing `webviewtracer-crawler/celery_workers/vv8_worker/uiharvester/execution_wrapper/application_runner/mode.py` line 180, the number of times an app can be retried can be changed by changing the last number at `webviewtracer-crawler/celery_workers/vv8_worker/entrypoint.sh`.

To use physical phones, the phones need to be connect to the computer using USB ports and the computer should have ADB installed. The python orchestration module should figure out the number of phones and assign each to it's own crawling module.

The `webviewtracer-crawler/raw_logs` contains all the logs for each app and `webviewtracer-crawler/crawl-data/` and the command `docker compose logs -f` can be used to obtain the logs for the traversals of each instance of emulator used.

D. Major Claims

The following are the claims being reproduced by the artifact, note that the claims are quantitative rather than numerical.

- (C1): SYSTEM discovers the leakage of context-restricted data from the app through injections in WebViews and is exfiltrated to the outside world.
- (C2): SYSTEM shows the presence of phantom exfiltrations, i.e. exfiltrations of data without associated injections.

E. Evaluation

The following experiments are being reproduced

1) Experiment (E1): Crawl [30 human-minutes + 16.5 compute-hours]: This experiment will run a small-scale crawl and provide numbers to prove C1 and C2.

Preparation and Execution Follow the steps at Section B

Results To fully reproduce our experiment, the user should see some apps loading WebViews and subsequently using WebViews to exfiltrate context restricted information. The results command (`python3 ./scripts/wvt-cli.py results`) should provide a table of the kinds of context restricted information being exfiltrated out. Some apps might load test ads due to non-standard nature of the emulator in which case there might be lesser injections and exfiltrations of context-restricted information. There should at least a few categories where the exfiltrations is very low compared to those reported by our paper. This is expected since data like "City" and IP information will vary from area to area. We provide an existing list of regexes at `webviewtracer-crawler/scripts/results.py` lines 8-26 which can be modified and swapped out entirely based on the environment the reviewer is using. We encourage the users to try and swap out the regex and try their own ones to test the system! The regexes we used are documented in `webviewtracer-crawler/scripts/experiment_pii_regexes.py`