Reproducibility Submission (Paper ID #80)

README for reproducibility submission of paper ID #80 ("Recurring Verification of Interaction Authenticity Within Bluetooth Networks") by Peters et al. @ WiSec'21.

The VM can be downloaded from Google Drive.

The VM was created using Vagrant. The default login is:
- **username:** vagrant
- **password:** vagrant

The files for this release are stored in `home/vagrant/via`.

I recommend using `xterm` for a shell on this VM. *(This version of Ubuntu seems to have issues with Terminal and higher Python versions.)*

A) Source Code Info

All of the necessary code should be installed on the provided VM already. Some source code has been adapted for this VM release. *(These changes are not reflected in the GitHub repository.)*

- **List of Programming Languages:** Python, Jupyter Notebook
- **Compiler Info:** Python 3.8.10 *(Running within a Python Virtual Environment)*
- **Packages/Libraries Needed:** See `requirements.txt`, which lists Python dependencies and their respective versions. To parse the Bluetooth traces (See Section B), we make use of extensions to two open source projects:
  - https://github.com/traviswpeters/btsnoop.git
  - https://github.com/traviswpeters/bluepy.git

B) Datasets Info

Repository: https://doi.org/10.15783/tjt0-b278

The dataset is being prepared to be officially released through the CRAWDAD project. For convenience, the raw data is pre-installed on the VM.

Overview

This dataset consists of a collection of **Bluetooth HCI traces** captured on a smartphone while a smartphone and smart device communicated. See `testbed.pdf` for an overview of the smartphone apps and the smart devices.

- `data/00_raw/` contains the raw HCI traces (btsnoop files pulled from an Android smartphone) - each subfolder contains the traces captured during communication between a specific device and its companion smartphone app.
- `data/01_processed/` contains CSV-formatted files, which are parsed versions of the raw Bluetooth traces. The first row of each file contains the column labels. *(This directory can be deleted/re-generated using `setup.sh` or the provided `Makefile`.)*

This dataset provided here includes both the raw traces as well as parsed versions of the files to make analysis easier (e.g., if you are not familiar with parsing Bluetooth/BLE packets.) If you have your own Bluetooth protocol parser, the raw files should be sufficient. Otherwise, you may prefer to work with the processed files.

To parse HCI traces, we extended two open-source projects: `btsnoop` and `bluepy`. Our extensions extend the parsing of the HCI protocol and other protocols that the HCI protocol encapsulates; namely, we extract

features for each packet within the HCI traces, such as packet types, lengths, endpoint identifiers, protocol semantics, and segmented headers and payloads belonging to higher-level Bluetooth protocols (e.g., the Attribute Protocol (ATT), the Security Management Protocol (SMP), the Signaling Protocol); and, because each trace captured a single app-device session, we labeled each packet according to the device it was sent to/from. These features and labels were written to CSV-formatted files for subsequent analysis.

C) Hardware Info

Network Configuration

Bluetooth networks are ad hoc networks. The dataset was collected by connecting various Bluetooth devices with a smartphone (i.e., a smartphone ad hoc network or "SPAN") and capturing communication between the devices and their companion app running atop the smartphone.

Collection Methodology

At the time of our experiments that were reported in our paper, the pre-processing code was executed on a VirtualBox VM running Linux, kernel version 4.14. The VM was configured to have 2 processors with 2048 MB base memory.

Our packaged VM runs kernel version 4.15; our experiments appear to be stable on this VM as well.

The host machine was a 2019 MacBook Pro.

```
Model Name:                 MacBook Pro
Model Identifier:           MacBookPro16,1
Processor Name:             8-Core Intel Core i9
Processor Speed:            2.3 GHz
Number of Processors:       1
Total Number of Cores:      8
L2 Cache (per Core):        256 KB
L3 Cache:                   16 MB
Hyper-Threading Technology: Enabled
Memory:                     16 GB
```

Data pre-processing is necessary to generate an intermediate file with per-packet features and labels.

The smartphone apps used in data collection were installed on a Nexus 5 smartphone running Android 6.0.1 ("Marshmallow"), API level 23, kernel version 3.4.0. Along with executing the apps, the smartphone also served as our primary device for data collection. To capture HCI traces, we enabled the **Bluetooth HCI snoop log** developer option. (This feature is a common developer option introduced in Android 4.4. It is interesting to note that using this feature does not even require rooting the phone.) The HCI snoop log captures all Bluetooth HCI packets to a binary-encoded file, which it writes to an SD card; the log format resembles the Snoop Version 2 Packet Capture File Format described in RFC 1761 ("btsnoop").

Each trace captured interactions between one app-device pair. Specifically, each trace captured all communications observed at the HCI layer (and therefore all protocol layers above the HCI layer). For each app-device pair we collected at least 10 traces, each of which included 3-10 minutes of network activity.

We gathered HCI traces by manually using the apps and devices in our testbed to emulate a wide variety of normal app-device interactions. The actions we performed consisted of: navigating the "official" smartphone app and exercising features that trigger network communication with a corresponding device, as well as acting upon the devices in such a way that triggers communication with its corresponding smartphone app.

After collecting each HCI trace (a btsnoop file), we moved the raw file from the smartphone to a local VM using the Android Debugger command line tool (adb). We reset the HCI snoop file on the smartphone

between each trace so that each HCI snoop file contained only packets belonging to interactions between a particular app-device combination; we refer to this as an app-device session.

Methodology Limitations

It was not our intention to discover and exercise every functional feature (and thus every BLE service or characteristic) of a particular app/device. Rather, it was our intention to observe typical features and interactions between devices and their official app, which could be used to construct normality models suitable for performing verification in future app-device interactions.

D) Experimentation Info

D1) Scripts and how-tos to generate all necessary data or locate datasets

D2) Scripts and how-tos to prepare the software for system

Run `./setup.sh`

> To avoid confusion, `setup.sh` exists to ensure that necessary dependencies are installed (D2) *before* processing the raw data (D1). All of these steps ***must*** be run before you can use the `runExperiments.sh` script.

D3) Scripts and how-tos for all experiments executed for the paper

Run `./runExperiments.sh`

> The original tables and figures were generated within a Jupyter Notebook. I exported the notebook and adapted the code to be structured in the required format, as described here: https://sites.nyuad.nyu.edu/wisec21/replicability-label/ Because "cell magic" functions are used, the best way I've found to run these standalone/exported scripts is to use `ipython` to invoke the scripts. See examples below.

D4) Clean-up

A Makefile is provided with a few helpful targets if you want to clean up the figs/tables, re-run experiments, or even clean up all of the processed data.

```
# clean up
make clean # or `make cleanhard`
```