Recolección de evidencia digital sobre dispositivos móviles

Informe ejecutivo

Autores
Juan Andrés Diana Robledo
José Ignacio Varela Gallino

Supervisores
Gustavo Betarte
Marcelo Rodríguez

Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
2015

Resumen

El área de *mobile forensics* hoy en día tiene un papel importante en la informática forense debido al extenso uso que tienen los dispositivos móviles. Características como la fabricación de los mismos utilizando arquitecturas SoC (*System on a Chip*) y la diversidad que presenta el ecosistema móvil, traen una serie de problemáticas desde el punto de vista forense.

Para realizar la extracción de datos de estos dispositivos, los investigadores forenses deben contar con una gran cantidad de herramientas. Cada una de estas, además de implicar un costo económico y de aprendizaje significativo, a menudo suelen utilizar formatos propietarios para expresar sus resultados, lo cual dificulta la reutilización de los mismos.

El presente trabajo realiza un estudio del estado del arte de la extracción de datos de dispositivos móviles en el marco del proceso forense informático y profundiza sobre las diversas problemáticas que se encuentran al realizar esta tarea, particularmente en la plataforma Android.

Para solucionar varios aspectos de estos problemas, desarrollamos una herramienta que busca facilitar la extracción de datos de dispositivos móviles Android. La misma brinda un nivel de abstracción que permite al usuario especificar los datos que desea obtener sin necesidad de tener que saber cómo es realizada la extracción de los mismos. Además, ésta utiliza el lenguaje estándar y abierto CybOX (*Cyber Observable eXpression*) para representar la información resultante de la extracción de datos.

Por último, la herramienta cuenta con un diseño que permite la extensibilidad de la misma con el fin de soportar tanto nuevas formas de extraer datos como nuevos dominios de datos. Esto brinda la posibilidad a un desarrollo colaborativo del conocimiento, de forma que el mismo pueda ser compartido para escalar ante la rápida evolución del ecosistema móvil.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Con la inmensa adopción que han tenido los dispositivos móviles durante esta última década, el área de *mobile forensics* ha pasado a tener un papel protagónico en la informática forense. Si bien podemos encontrar en esta área muchas similitudes con la de *computer forensics*, debido a varias de las características que poseen los dispositivos móviles y a su rápida evolución, los mismos presentan una serie de desafíos importantes desde el punto de vista forense.

Aunque el área de *mobile forensics* pasó a ocupar un lugar importante dentro de la informática forense tanto en la industria como en el ambiente académico, la misma es aún relativamente nueva comparada con *computer forensics* que ya cuenta con varias décadas más de madurez. Esto queda en evidencia si observamos los estándares sobre procedimientos forenses que podemos encontrar hoy en día para dispositivos móviles.

Una etapa fundamental que forma parte de cualquier proceso forense informático es la de obtener datos de un dispositivo. En este sentido, los dispositivos móviles presentan varios desafíos para llevar a cabo esta etapa de manera adecuada. En el Capítulo 2 estudiaremos en detalle varios de ellos. En particular, veremos por qué no es una tarea sencilla obtener una copia bit-a-bit del medio de almacenamiento primario de un dispositivo móvil como lo suele ser en el caso de una PC.

La diversidad con que son diseñados los dispositivos móviles y las diversas interfaces de hardware y software que presentan son factores que juegan en contra para lograr contar con un proceso de obtención de datos uniforme para los dispositivos móviles. Esto lleva a que los investigadores forense deban a menudo poseer una multitud de herramientas para extraer los datos de los diversos dispositivos móviles a los que se enfrentan. El costo de las herramientas y el aprendizaje que conlleva cada una de ellas imponen un obstáculo significativo para muchas organizaciones.

Si bien el tipo de información que deseamos extraer de los dispositivos móviles usualmente es el mismo a lo largo de cualquier fabricante o sistema operativo, el formato en que dicha información se encuentra o la forma que debemos utilizar para acceder a ella pueden diferir.

Si consideramos la plataforma móvil Android, que cuenta con la mayor porción del mercado a nivel mundial (más del 80 % según la firma Gartner [17]), la diversidad en sus diseños es un hecho notorio tanto en software como hardware. Para tener una noción de la magnitud de la diversidad que presenta el ecosistema Android, observemos que en agosto de 2015 se podían encontrar más de 24.000 modelos de dispositivos Android producidos por más de 1.000 fabricantes distintos [34].

De esta forma, vemos que si pudiéramos contar con un nivel de abstracción que nos permitiera concentrarnos en qué datos deseamos obtener, sin tener que estar considerando cómo debemos realizar la extracción de los mismos de cada dispositivo en particular, estaríamos dando un paso muy importante hacia el desarrollo de herramientas y procesos forenses que escalen ante la creciente variedad de dispositivos móviles.

1.2. Objetivos

En una primera instancia, el objetivo que se planteó fue realizar una recopilación del estado del arte en lo que refiere a extracción de datos de dispositivos móviles. La misma nos permitió comprender las distintas problemáticas que presenta hoy en día llevar a cabo esta tarea. De esta forma, luego pudimos identificar diversas oportunidades interesantes para investigar en mayor profundidad.

En una segunda instancia y en base al trabajo de estado del arte elaborado, buscamos abordar la problemática descrita en la sección anterior con el obje-

tivo de dar solución a varios de los problemas planteados mediante el diseño de una herramienta que facilitara la extracción de datos, fuera extensible y utilizara un lenguaje estándar para expresar sus salidas.

1.3. Enfoque seguido

Este trabajo acompaña a la línea de investigación que viene desarrollando en el área de informática forense el grupo de seguridad informática (GSI) de la Facultad de Ingeniería de la República Oriental del Uruguay. El mismo, que como vimos se enfoca a la extracción de datos de dispositivos móviles, complementa a un trabajo dedicado al análisis forense de estos datos que fue realizado de forma simultánea a este por otros estudiantes de grado.

Para la investigación del estado del arte sobre el tema, partimos de los lineamientos que ofrece el NIST sobre cómo trabajar con dispositivos móviles en el ámbito forense. Esto nos brindó una perspectiva completa de las diversas etapas del proceso forense en el que se encuentra inmerso el proceso de extracción de datos. Luego, investigamos varios de los tipos de métodos de extracción que existen y su aplicados a dispositivos móviles, en donde terminamos tomando especial interés por los tipos de extracciones lógicas. Posteriormente, realizamos una extensa investigación de los formatos disponibles para almacenar datos extraídos así también como lenguajes para representar información sobre los mismos. Finalmente, decidimos tomar a la plataforma móvil Android para profundizar y aplicar los temas estudiados.

Para el diseño de la herramienta buscamos cumplir con los tres objetivos mencionados en la sección anterior. La misma fue concebida para ser utilizada por investigadores forenses sin necesidad de poseer el conocimiento técnico específico de cómo se realiza la extracción y examinación de los datos. Asimismo, se hizo especial énfasis en el diseño de su arquitectura con la finalidad de permitir la extensibilidad de varios de sus componentes fundamentales como son operaciones, tipos de datos y tipos de fuentes de datos. Por último, para representar información sobre los datos extraídos la herramienta hace uso del lenguaje CybOX desarrollado por MITRE y que recientemente acaba de pasar a manos de OASIS (*Organization for the Advancement of Structured Information Standards*) para continuar su desarrollo como estándar abierto.

En cuanto al proceso de desarrollo utilizado para implementar el prototipo de la herramienta, tomamos un enfoque iterativo e incremental. El mismo estuvo acompañado por el desarrollo de un conjunto de pruebas automáticas

con el fin de ir verificando las funcionalidades desarrolladas en cada etapa y permitir una rápida sucesión de iteraciones al reducir la cantidad de defectos introducidos en cada iteración.

Por último, el caso de estudio realizado tiene la finalidad de validar el prototipo de la herramienta construida para una serie de escenarios similares al uso real que se le daría y además mostrar varios de los aspectos claves del desarrollo de extensiones de la misma.

1.4. Estructura del documento

A continuación describiremos el contenido del resto del documento de forma de brindar una noción de alto nivel del documento.

El Capítulo 2 presenta un resumen del trabajo de investigación de estado del arte que realizamos. A menos que el lector ya se encuentre familiarizado con las temáticas tratadas (en especial el lenguaje CybOX y la plataforma Android), este capítulo sienta las bases que permitirán comprender el trabajo.

El Capítulo 3 plantea y analiza el problema a resolver. Se muestra el enfoque inicial tomado para encarar el problema y se describen los conceptos fundamentales. Luego, se establecen los requerimientos, los casos de uso considerados y el alcance del trabajo. Finalmente, se analizan las decisiones por las cuales se escogió el lenguaje CybOX para representar los datos de salida.

El Capítulo 4 presenta el diseño de la herramienta propuesta para resolver el problema. De esta forma, se describe el modelo de datos utilizado y la arquitectura propuesta. Además, se ven en detalle los puntos de extensibilidad con que la misma deberá contar.

El Capítulo 5 describe las decisiones más importantes que tomamos al realizar la implementación del prototipo de la herramienta así también como las herramientas y el ambiente utilizado para llevar a cabo la misma.

El Capítulo 6 muestra varias de las funcionalidades claves de la herramienta a través del desarrollo de diversas extensiones y la utilización de un conjunto de datos que pone a prueba a la herramienta frente a un caso semejante a uno real.

El Capítulo 7 plantea una serie de oportunidades interesantes para continuar trabajando en el desarrollo de la herramienta.

El Capítulo 8 presenta las conclusiones obtenidas a partir del trabajo realizado.

Por último, el documento cuenta con dos apéndices. El Apéndice A proporciona una guía que resulta imprescindible en caso que el lector tenga interés en desarrollar extensiones para la herramienta. Mientras que el Apéndice B contiene los diagramas de flujo de interacción entre los componentes del sistema para cada uno de los comandos de la herramienta.

Capítulo 2

Estado del Arte

En este capítulo presentaremos de manera resumida el trabajo realizado sobre el estado del arte de la recolección de datos de dispositivos móviles. En la última sección profundizaremos su estudio tomando el caso particular de la plataforma Android. En [22] se encuentra el trabajo completo en caso que se desee profundizar en los temas a ver y otros relacionados.

Los temas que no se encuentran directamente relacionados con el presente trabajo fueron omitido del resumen. Entre ellos podemos destacar: SIM cards, redes inalámbricas, tecnologías de almacenamiento flash, formatos de almacenamiento de los datos extraídos, aspectos del almacenamiento de dispositivos Android (en particular, particiones y filesystems habitualmente utilizados en Android), formas de obtener root en dispositivos Android y finalmente el lenguaje DFXML. La mayor parte de los temas mencionados se encuentran íntimamente ligados a lo que conoceremos en la sección 2.3 como extracciones físicas, las cuales no forman parte del alcance de nuestro trabajo.

2.1. Características de los dispositivos móviles

Empezaremos por conocer los aspectos fundamentales de este tipo de dispositivos que serán nuestro objeto de estudio. Entender sus características particulares es de suma importancia si pretendemos conocer cómo extraer información de los mismos y entender qué consecuencias tienen nuestras acciones a lo largo de este proceso.

En el área forense, contamos con un desarrollo del conocimiento sobre compu-

tadoras personales (PC) muy avanzado. El mismo ha sido posible debido a que si bien en las últimas décadas las capacidades de las PC han estado en constante incremento, las bases de su funcionamiento e interfaces brindadas no han presentado cambios radicales. Esto ha permitido que hoy en día en las PC, al utilizar determinadas metodologías y procedimientos, podamos contar con ciertas garantías sobre los datos extraídos que resultan muy importantes del punto de vista forense.

Las PC son los sistemas informáticos que presentan una mayor semejanza a los dispositivos móviles de hoy en día, debido a la gran cantidad de aspectos que comparten. Por lo tanto, resulta razonable que intentemos aplicar el conocimiento con el que contamos en el área forense sobre las PC, también a los dispositivos móviles. Esto es posible en ciertos aspectos en los que comparten las mismas características con las PC. Sin embargo, veremos que los dispositivos móviles presentan una serie de características muy diferentes, las cuales implican que a menudo no sea posible aplicar las mismas técnicas y debamos buscar nuevas alternativas.

Además de introducir nuevos desafíos desde el punto de vista forense, los dispositivos móviles también brindan nuevas oportunidades al contar con ciertos tipos de información a los cuales no estábamos acostumbrados a encontrar en las PC.

2.1.1. Aspectos de hardware

La cantidad de modelos de dispositivos móviles que existen y su variedad están en continuo aumento. Como se mencionó en el capítulo 1, si consideramos únicamente aquellos dispositivos móviles que corren el sistema operativo Android, podemos observar que en agosto de 2015 existían más de 24.000 modelos distintos.

¿Por qué resulta relevante considerar la inmensa cantidad de modelos de dispositivos móviles que existen, si después de todo, podemos observar que en el terreno de las PC también existe una gran variedad de modelos y sin embargo las herramientas forenses con las que contamos hoy en día no presentan mayores dificultades en soportar esta diversidad?

Resulta que los dispositivos móviles presentan, en la actualidad, una particularidad muy importante que hace que la gran diversidad de modelos sea un hecho relevante. Se trata de que presentan una gran variedad de interfaces de hardware, incluso entre modelos de un mismo fabricante. Esto se ve

acentuado por el hecho de que muy a menudo los dispositivos móviles cuentan con hardware especializado. Esta diversidad de interfaces se extiende y probablemente es más visible al público en general al observar la cantidad de conectores de cables de poder y datos que existen para dispositivos móviles. La Unión Europea incluso intentó impulsar regulaciones con el objetivo de obligar a los fabricantes de estos dispositivos a adoptar un estándar único para los conectores de teléfonos celulares [21].

La diferencia a apreciar en la industria de las PC es que el hardware es diseñado con una visión modular para proveer compatibilidad con otras piezas de hardware construidas potencialmente por otros fabricantes. Por ejemplo, en caso de querer acceder a los datos de un disco duro de un PC, contamos con un conjunto de interfaces estándar muy limitado en la industria (IDE, SATA, SCSI).

Si consideramos ahora a los dispositivos móviles, vemos que proveer interoperabilidad entre componentes de hardware de diversos fabricantes no es un objetivo que suelen tener en mente los fabricantes a la hora de diseñar un nuevo dispositivo. Frecuentemente priorizan el poder contar con un diseño compacto, muchas veces soldando los componentes directamente o diseñando chips que integran varias funciones con el objetivo de disminuir el espacio. Esto se evidencia en las arquitecturas SoC (System on a Chip), populares en el mundo móvil. Muchos dispositivos hoy en día vienen incluso con su batería soldada al PCB por esta misma razón.

Mientras en la industria la tendencia de dispositivos móviles con diseños monolíticos se acentúa, un grupo de investigación de Google (Advanced Technology and Projects group) ha tenido la iniciativa, en estos últimos años, de formar la base para contar con dispositivos móviles modulares. Bajo el nombre de “Project Ara”, proponen una plataforma que brinde el esqueleto mínimo necesario de interfaces estándar con el objetivo de permitir que fabricantes de diversos tipos de piezas de hardware de dispositivos móviles (como cámara, radio, batería, almacenamiento, etc) puedan diseñarlas de forma que interoperen con facilidad con componentes de otros fabricantes. El usuario del dispositivo entonces tendría la libertad de escoger componentes de diferentes fabricantes según sus necesidades y armar su dispositivo a medida. El proyecto se encuentra en una etapa temprana con prototipos avanzados e interés de varios fabricantes [20].

Es interesante contemplar por un instante esta iniciativa ya que su éxito podría tener una repercusión significativa en el proceso de adquisición de

datos. El hecho de contar con interfaces estándar de hardware podría facilitar en gran medida el proceso. Además, brindaría la habilidad de poder remover físicamente los módulos de hardware con facilidad. Esta última es una diferencia notoria que presentan hoy en día los dispositivos móviles. En particular, cuando consideramos el acceso a datos almacenados, a diferencia de las PC, la unidad de almacenamiento principal se encuentra soldada al PCB del dispositivo. La única forma de remover dicha unidad es mediante técnicas de "chip off" como veremos más adelante en la sección 2.3. La excepción a esto son las tarjetas de expansión (SD cards), las cuales se encuentran presentes en muchos modelos. Veremos más adelante que si bien son una fuente de información importante, los datos más sensibles suelen estar en el almacenamiento principal.

2.1.2. Aspectos de software

En esta sección nuestra intención es ver en más detalle los aspectos que caracterizan al software de los dispositivos móviles.

2.1.2.1. Sistema operativo

Hoy en día, todos los sistemas operativos para dispositivos móviles cuentan con capacidades comparables a los tradicionales sistema operativos de PC. Sin embargo, además de presentar grandes cambios en cuanto al diseño de interacción con el usuario, estos sistemas también presentan varios aspectos técnicos importantes que los diferencian de los sistemas anteriores. Veamos algunos de estos aspectos que son de interés:

1. Uso eficiente de los recursos limitados con los que cuentan.
   - **Poder de procesamiento**: Si bien los dispositivos móviles cada vez cuentan con un mayor poder de procesamiento, la brecha entre el procesamiento con el que cuentan los PCs hoy en día aún es considerable.
   - **Almacenamiento**: De forma similar, la capacidad de almacenamiento ha aumentado considerablemente, pero es significativamente menor con la que podemos contar en las PCs, por lo cual tanto aplicaciones como sistema operativo debe hacer un uso eficiente del mismo.
   - **Batería**: El ritmo con que avanza la tecnología informática no ha tenido su contrapartida en la tecnología de las baterías. La dura-

ción de las mismas no ha aumentado considerablemente. Por lo tanto, el mismo pasa a ser un recurso de mayor relevancia. Para aumentar la duración de la batería se debe intentar ser lo más eficiente posible con todos los recursos del dispositivo en general. Desde el punto de vista del software, esto quiere decir tener mecanismos dentro del sistema operativo como pueden ser disminuir el brillo de la pantalla en ciertas situaciones, matar procesos que hagan uso extenso de los recursos, etc. También implica que los desarrolladores de software deben tener especial cuidado al implementar funcionalidades en sus aplicaciones.

2. Los sistemas operativos móviles introducen varios conceptos de seguridad importantes a sus modelo de seguridad con respecto a los disponibles por defecto en sistemas operativos de PC. Veamos dos muy importantes que se encuentran en los dos sistemas más populares (Android e iOS):
   - **Sandboxing de aplicaciones**: El sistema operativo encapsula a cada aplicación imponiendo una capa a través de la cual la aplicación debe interactuar con el sistema operativo pasando controles antes de acceder a recursos fuera de la misma.
   - **Control de acceso a recursos**: Establecen un fuerte control de acceso a recursos basado en permisos. Mediante el mismo buscan comunicar de forma clara al usuario los recursos a los cuales tiene acceso cada aplicación.

3. El ciclo de liberación de versiones de los sistemas operativos móviles es considerablemente más rápido que al que estábamos acostumbrados para sistemas operativos de PC. iOS tiene un promedio de una nueva versión por año [44] mientras que para Android el promedio es menor a un año [42]. En los sistemas operativos de escritorio el promedio de tiempo entre nuevas versiones rondaba los cuatro años [43].

4. Por último, como hemos visto, existe una mayor diversidad de sistemas operativos y cambios en las cuotas de mercado de los mismos que en la industria de las PC. Hasta hace unos años, se dió una competencia muy intensa entre varios contendientes con el fin de capturar el mercado. En ese período surgieron muchos sistemas operativos móviles. Hoy en día tenemos dos claros vencedores: Android e iOS [17]. De todas formas, continúan surgiendo sistemas operativos móviles como es el caso de Windows Phone (de Microsoft), Tizen (de Samsung), Firefox OS (de Mozilla) y más. Por otro lado, otros que tuvieron gran

participación del mercado en el pasado reciente como Blackberry OS, Symbian OS, etc han ido desapareciendo lentamente. Estos cambios en un lapso tan corto le da un aspecto de gran dinamismo a la industria de los dispositivos móviles. Para herramientas que dependen de estas plataformas, esto significa que las mismas en general deben estar en constante actualización.

2.1.2.2. Aplicaciones

Empecemos por recordar que la distribución de aplicaciones en PC tradicionalmente se ha dado de forma descentralizada y los controles que se realizan antes de instalar una aplicación de escritorio suelen ser mínimos por parte del sistema operativo. Veamos cómo el ecosistema de aplicaciones que existe en el mundo móvil es notoriamente distinto.

En el modelo utilizado usualmente por las plataformas móviles se cuenta con una tienda virtual que centraliza el descubrimiento y distribución de aplicaciones. Esto además viene acompañado de un mecanismo que facilita al usuario la instalación y actualización de las mismas. Esta capacidad centralizada le brinda varias ventajas importantes a la organización responsable de la plataforma.

Por un lado, tiene la capacidad de controlar varios aspectos del proceso, entre ellos, cómo los usuarios descubren las aplicaciones, qué tipo de contenido consideran apropiado distribuir, para cuáles regiones geográficas desean que esté disponible cierto contenido, y muchos más. Además, pueden imponer ciertos requerimientos de calidad sobre las aplicaciones.

Por otro lado, tiene la capacidad de implantar mecanismos que le permitan mitigar la distribución de malware y reducir de esta forma significativamente el impacto en los usuarios de la plataforma.

2.2. Diversidad de los tipos de datos

A pesar de las dificultades que suelen presentar los dispositivos móviles a la hora de extraer datos, estos son una excelente fuente de datos ya que cuentan con una gran cantidad de información, mucha de la cual no solemos encontrar en otros tipos de dispositivos como PCs.

Su naturaleza personal hace que podamos establecer una relación de uno

a uno entre el dispositivo y su propietario. Esto facilita la atribución de acciones a un individuo. En investigaciones forenses dicha relación resulta clave al permitir establecer la última milla de evidencia que la relaciona a un individuo [16].

Hoy en día ya no contamos con un conjunto de tipos de datos finitos y previsibles como era usual al examinar feature phones los cuales contaban con unos pocos tipos de datos como contactos, mensajes, fechas de calendario, etc. Mediante el modelo adoptado por las plataformas móviles, los dispositivos cuentan con acceso de forma muy sencilla a una inmensa cantidad de aplicaciones para una gran diversidad de usos. Esto hace que podamos llegar a encontrarnos con tipos de datos de toda índole.

De esta forma, tendremos que estar preparados para enfrentarnos a una gran diversidad de tipos de datos. Con el fin de tener una noción de los tipos de datos con los que vamos a trabajar, veamos un conjunto representativo de las grandes categorías de los tipos de datos encontrados en dispositivos móviles que suelen ser de interés en investigaciones forenses:

- Telefonía
- PIM (Personal Information Management)
- Mensajería
- Redes sociales
- Geolocalización
- Metadatos
- Sistema operativo
- Servicio celular

En el documento completo de Estado del Arte en [22], el lector podrá encontrar varios ejemplos de tipos de datos para cada categoría y ejemplos de los usos que se le pueden dar en el contexto de una investigación forense.

2.3. Tipos de métodos de extracción

A la hora de extraer los datos de un dispositivo móvil es necesario tener en cuenta las características del método de extracción a utilizar. Estas características hacen referencia tanto a las capacidades de los métodos a la hora de

obtener los datos como a las consecuencias que los mismos tengan sobre el dispositivo [30].

En la siguiente figura se puede ver los distintos tipos de métodos de extracción:

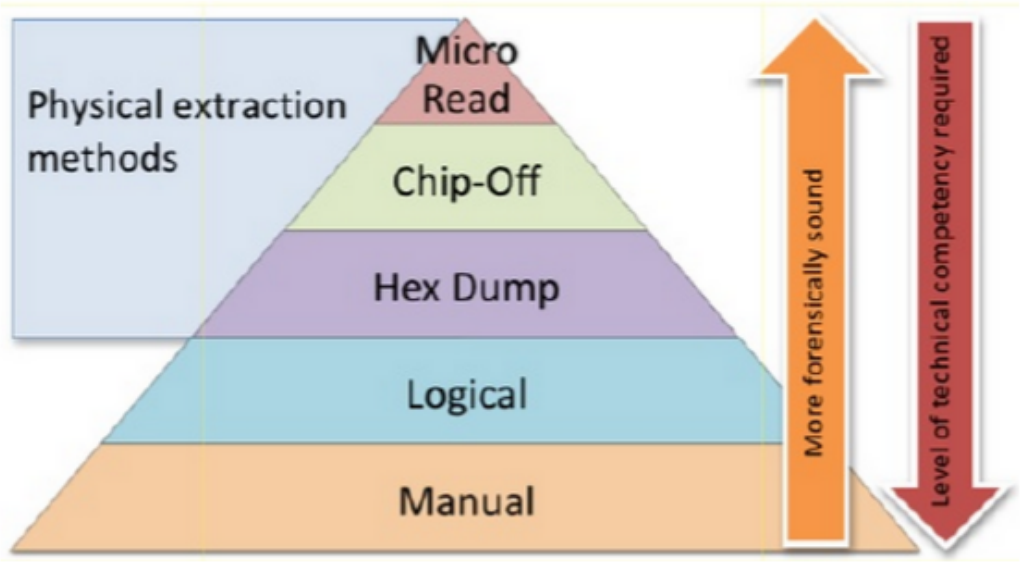

Figura 2.1: Tipos de métodos de extracción [11]

A medida que nos movemos hacia la cima de la pirámide, los métodos:

- Resultan más limpios desde el punto de vista forense.
- Requieren de utilizar herramientas más caras.
- Requieren mayor conocimiento técnico.
- Exigen más tiempo para analizar los datos extraídos.
- Requieren más entrenamiento.
- Son más invasivos.

Comenzando por la base de la pirámide, los métodos *manuales* requieren interactuar con el dispositivo utilizando el teclado o pantalla táctil del mismo para obtener los datos. Esto hace que este tipo de métodos sea soportado por una gran cantidad de dispositivos. Dado que la interacción con el dispositivo

se da a través del sistema operativo, únicamente se pueden obtener datos que estén disponibles a través del mismo [16].

En el segundo nivel se encuentran los métodos de extracción *lógicos*, los cuales consisten en conectar el dispositivo a una PC y utilizar algún protocolo de intercambio de datos para obtener los datos del dispositivo. Al igual que con los métodos manuales, los únicos datos que se pueden obtener son los accesibles a través del sistema operativo. Son métodos rápidos, automatizables y la interpretación de los datos obtenidos cuenta con las abstracciones del sistema operativo.

En el tercer nivel tenemos los métodos de extracción *físicos*. Estos consisten en copiar de forma parcial o total los datos del dispositivo en su forma bruta (raw data). A diferencia de los métodos manuales y lógicos, los métodos físicos no interactúan con el sistema operativo del dispositivo. Debido a esto, se puede obtener la totalidad de los datos del dispositivo (incluyendo datos borrados), pero hay que tener en cuenta que estos datos deberán ser interpretados.

En el cuarto nivel se encuentran los métodos conocidos como *chip-off*. Como lo dice su nombre, estos métodos consisten en remover físicamente el chip de memoria del dispositivo, y luego (utilizando equipos especializados) realizar la lectura del contenido del chip de forma directa. Al igual que los métodos del nivel anterior, estos métodos permiten obtener la totalidad de los datos del dispositivo. Los aspectos negativos más importantes son el hecho de que el dispositivo queda inoperante y los datos obtenidos deben ser interpretados.

En el último nivel se encuentran los métodos *micro read*. En estos métodos se utilizan microscopios poderosos para ver el estado de la memoria del dispositivo, siendo los mismos los más difíciles y caros de llevar a cabo.

2.4. Lenguajes de representación de los datos

Tras tener los datos extraídos, comienza la tarea de organizar la información relevante que encontremos en los mismos. Para esto, se suelen utilizar lenguajes mediante los cuales podamos representar los datos relevantes que fueron adquiridos e información sobre el proceso utilizado para extraerlos.

De esta forma, estos lenguajes nos permiten expresar y compartir datos relevantes de una investigación. Esto puede resultar de inmensa importancia

cuando se requiere trabajar con varios equipos dado que facilita en gran medida la colaboración. Además, nos permite automatizar procesos, y en la medida que el lenguaje adquiera adopción, su interoperabilidad entre herramientas resultará más sencilla.

Es importante que tengamos en cuenta que la elección del lenguaje para representar los datos examinados limitará las herramientas con las que podremos trabajar con facilidad. Uno de los inconvenientes presentes hoy en día en la comunidad forense es el hecho de que cada herramienta comercial utiliza sus propios formatos propietarios y esto hace que la interoperabilidad entre herramientas sea difícil. Por esto, resultan de gran importancia las iniciativas que están surgiendo de formatos abiertos y los esfuerzos de estandarización [33].

En los últimos años han surgido varios lenguajes de representación de datos de evidencia forense que son interesantes. En particular, nosotros consideramos dos (CybOX y DFXML) para su uso en el trabajo. Ambos son lenguajes abiertos y que poseen un desarrollo activo del mismo y sus herramientas.

En [2] se realiza un estudio comparativo de las diversas alternativas de lenguajes de intercambio que se podían encontrar en 2010. Entre ellos, se describen los siguientes dos: DEX (Digital Evidence Exchange) [10] y XIRAF [41], los cuales plantean ideas interesantes pero los que no llegamos a considerar debido a que en el primer caso el lenguaje no cuenta con un desarrollo activo desde hace años y en el segundo caso tanto el lenguaje como el framework que lo soporta no se encuentran disponibles públicamente.

Por lo tanto, nuestras alternativas a considerar fueron DFXML y CybOX. Nos inclinamos por CybOX no sólo debido a la gran comunidad activa que posee, sino también porque permite representar prácticamente todos los datos que permite representar DFXML y por el hecho de que DFXML está diseñado para trabajar únicamente con extracciones físicas. A continuación veremos los aspectos que consideramos importantes conocer sobre CybOX para luego comprender su uso en este trabajo.

2.4.1. CybOX

CybOX [28] es un lenguaje estructurado para representar lo que se denominan *cyber observables*. Estos son eventos o propiedades con estado que pueden ser observados en el dominio cibernético. El lenguaje no busca satisfacer un caso de uso en particular, sino que intenta ser lo suficientemente flexible

para prestar una solución común a un amplio espectro de casos de uso de ciberseguridad que requieran manejar cyber observables.

CybOX soporta un amplio espectro de dominios de ciberseguridad incluyendo caracterización de amenazas, caracterización de malware, manejo operacional de eventos, respuesta de incidentes, intercambio de indicadores de compromiso, digital forensics y más.

De esta forma, vemos que CybOX es utilizado para una gran diversidad de finalidades. En cada una de estas el mismo generalmente es utilizado de forma indirecta, esto es, a través de otro lenguaje que aprovecha su expresividad para describir cyber observables en su dominio.

2.4.1.1. ¿Qué intenta resolver?

Podemos considerar tres principios que caracterizan las intenciones del lenguaje [26]:

1. **No está dirigido a un caso de uso de ciberseguridad en particular**: La intención es que sea lo suficientemente flexible para ofrecer una solución común para todos los casos de uso de ciberseguridad que requieran la capacidad de trabajar con cyber observables. Por lo tanto, busca principalmente ser una infraestructura para representar esta información, sobre la cual puedan construirse otros estándares y soluciones en cada dominio de trabajo específico.
2. **Flexibilidad tanto para expresar instancias como potenciales patrones**: También busca ser lo suficientemente flexible para permitir describir tanto instancias de cyber observables de alto grado de fidelidad como patrones más abstractos de potenciales cyber observables.
3. **Visión de automatización integrada**: Al tener especificado un esquema común y estructurado para los cyber observables, es posible desarrollar la capacidad de automatizar el intercambio de información detallada de los mismos.

2.4.1.2. Componentes de CybOX

CybOX presenta un modelo de datos separado en cuatro componentes:

Core: Consiste en los esquemas CybOX Core y CybOX Common que definen

las construcciones básicas del lenguaje (como *Observable* y *Property*) y tipos de datos comunes utilizados por objetos (como *HashType* y *TimeType*).

Objects: Consiste de todos los esquemas de los objetos predefinidos que vienen por defecto con CybOX. Entre estos podemos encontrar Device, File, URI, User Account, etc.

Vocabularies: Consiste en varias listas de valores definidos que suelen resultar útiles para el uso por parte de los objetos. Por ejemplo, *HashNameEnum* contiene una lista de algoritmos de hashing como MD5, SHA256, etc. Es posible también crear uno mismo sus propios vocabularies.

Extensions: Consiste de esquemas externos que utiliza CybOX para representar algún aspecto como puede ser localización geográfica a través de CIQ o describir plataformas a través de CPE.

2.4.1.3. ¿Por qué y para qué nos sirve?

El lenguaje es de nuestro interés por las siguientes características que presenta:

- Su capacidad de representar propiedades con estado, hace que sea posible representar datos obtenidos en una extracción y metadatos de forma estructurada.
- El dominio de información sobre la cual trabaja incluye al de digital forensics. Muchos de los objetos predefinidos son relevantes a nuestro dominio y esto hace que podamos reutilizarlos.
- La visión de automatización y las diversas herramientas que proporciona para facilitar su procesamiento es un aspecto clave. Consideramos importante que la información producida en el lenguaje pueda ser utilizada por parte de otras herramientas con facilidad.
- El hecho de que sea un lenguaje estándar y el mismo sea utilizado por diversas comunidades es también importante. Indica la madurez del mismo y ayuda al soporte del mismo por parte otras herramientas.

2.4.1.4. Representación de tipos de datos

Como vimos, un aspecto interesante de CybOX es que el mismo nos brinda un conjunto de objetos predefinidos relevantes para el dominio de digital forensics. En su última versión a la fecha (2.1), podemos observar objetos como

File, Email Message, SMS Message, URI, etc que nos permiten representar varios tipos de datos de interés en el contexto del trabajo.

Por otro lado, cabe destacar que también es posible extender el conjunto de objetos que nos permite representar CybOX (esto es, más allá de los predefinidos). En la siguiente sección, veremos cómo podemos utilizar un mecanismo que brinda CybOX para hacer esto.

2.4.1.5. Utilizando CustomObject

El objeto *CustomObject* nos permite especificar objetos que no tienen un schema definido. Esto hace que resulte fácil utilizarlos para representar nuevos objetos pero para esto ambos productor y consumidor deben ponerse de acuerdo en los campos declarados dado que no cuenta con un schema que defina el nombre y tipo de cada uno.

Veamos un ejemplo en el cual tenemos un *CustomObject* [25] que representa la contraseña de un documento de Microsoft Office:

```
<CustomObj:CustomObjectType xsi:type="CustomObj:CustomObjectType"
    custom_name="example:OfficePassword">
  <cyboxCommon:Custom_Properties>
    <cyboxCommon:Property name="password"
    description="MS Office encryption password">
      SuP3rS3cr3T!
    </cyboxCommon:Property>
  </cyboxCommon:Custom_Properties>
    <CustomObj:Description>
      This is a string used as a password to protect an Microsoft Office
          document.
    </CustomObj:Description>
</CustomObj:CustomObjectType>
```

2.4.1.6. Extensión del lenguaje

Desde el punto de vista técnico, CybOX es un lenguaje que está basado en XML y se encuentra definido utilizando XML Schema. Esto implica que debemos contar con buen conocimiento de estos lenguajes si deseamos extender CybOX.

Existen varios puntos de extensión del lenguaje CybOX. En particular, los más frecuentes suelen ser la definición de nuevos objetos y vocabularios. Nosotros vamos a concentrarnos en el más interesante para nosotros que es extender el lenguaje mediante la definición de nuevos objetos, dado que nos

permitan representar tipos de datos que no son considerados por el lenguaje actualmente.

Creando un nuevo objeto de CybOX
Para crear un nuevo objeto en CybOX es necesario extender el tipo abstracto *ObjectPropertiesType* que se encuentra definido en CybOX Common. Hay dos formas ligeramente distintas de hacer esto:

- Extender directamente de *ObjectPropertiesType*. Básicamente vamos a estar creando un objeto totalmente nuevo. Esta construcción nos brinda la capacidad de definir las propiedades del objeto.
- Extender indirectamente de *ObjectPropertiesType*. Esto es, no extender de *ObjectPropertiesType* sino de otro objeto CybOX. Este otro objeto a su vez extiende, ya sea directa o indirectamente, a *ObjectPropertiesType* dado que todos los objetos deben extender a éste. Esta forma nos permite reutilizar objetos CybOX, evitando duplicar formas de expresar lo mismo y teniendo un modelo de datos más coherente. Por ejemplo, podemos tener un objeto de tipo *FileObjectType*. Si queremos ser más específicos, podemos optar por un objeto *ImageFileObjectType* que extiende a *FileObjectType* y agrega algunas propiedades como formato de imagen, dimensiones de la imágen, etc.

A su vez, resulta importante tener en cuenta los recursos disponibles con los que contamos en CybOX Common y CybOX Vocabularies. En particular, todos los tipos de datos simples que solemos contar en la mayor parte de los lenguajes ya se encuentran definidos (excepto booleano) en CybOX Common. Dentro de CybOX Vocabularies, podemos encontrar varias listas de valores que nos pueden resultar útiles como nombres de algoritmos, estándares de encoding de caracteres, etc.

La documentación de CybOX [28] provee una guía que describe a grandes rasgos los pasos que debemos tomar cuando vamos a crear un nuevo objeto:

1. Decidir qué es lo que queremos representar en CybOX.
2. Determinar qué campos/atributos pueden ser utilizados para caracterizar el objeto CybOX.
3. Establecer una correspondencia entre los tipos de datos de dichos campos y los tipos de propiedades disponibles en CybOX.
4. Revisar objetos CybOX existentes para ver si las capacidades que definimos en los pasos anteriores ya no se encuentran soportadas por un

objeto existente. Además, identificar objetos CybOX que soporten un subconjunto de las capacidades necesarias, de forma de evaluar extenderlo y cubrir las restantes.

5. Definir un namespace para el objeto.
6. Crear el esquema para el objeto.
7. Agregar documentación al esquema.

Ejemplo Veamos cómo podemos extender el Device Object de CybOX para que pueda representar un dispositivo más concreto: un teléfono celular. Por lo tanto, además de incluir descripción, fabricante, modelo, número de serie, versión de firmware, etc que ya incluye un Device Object, debemos agregarle IMSI, IMEI y ICCID. Aquí ya hicimos los primeros cuatro pasos.

A continuación debemos escoger un namespace para nuestro nuevo TelephoneDevice Object (`http://fing.edu.uy/objects#TelephoneDeviceObject-2`) y crear un archivo *Telephone_Device.xsd* que contenga el siguiente esquema:

```
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:cyboxCommon="http://cybox.mitre.org/common-2"
  xmlns:DeviceObj="http://cybox.mitre.org/objects#DeviceObject-2"
  xmlns:TelephoneDeviceObj="http://fing.edu.uy/objects#TelephoneDeviceObject-2"
  targetNamespace="http://fing.edu.uy/objects#TelephoneDeviceObject-2"
  version="1.0">
  <xs:import namespace="http://cybox.mitre.org/common-2"
      schemaLocation="../cybox_common.xsd"/>
  <xs:import namespace="http://cybox.mitre.org/objects#DeviceObject-2"
      schemaLocation="Device_Object.xsd"/>
  <xs:element name="Telephone_Device" type="DeviceObj:DeviceObjectType"/>

  <xs:complexType name="TelephoneDeviceObjectType">
    <xs:complexContent>
      <xs:extension base="DeviceObj:DeviceObjectType">
        <xs:sequence>
          <xs:element name="IMSI"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
          <xs:element name="IMEI"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
          <xs:element name="ICCID"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Si bien es bastante verbosa la representación, lo único que tuvimos que hacer conceptualmente fue extender el objeto Device de CybOX y agregarle las tres

propiedades que queríamos (IMSI, IMEI, ICCID) como strings.

Finalmente, para verificar que el esquema sea válido podemos verificarlo con la herramienta `xmllint` (disponibles en sistemas basados en Unix) de la siguiente forma:

```
xmllint --noout --dtdvalid http://www.w3.org/2001/XMLSchema.dtd
    Telephone_Device.xsd
```

2.5. Plataforma Android

En esta sección estudiaremos las características más importantes a tener en cuenta para extraer datos de un dispositivo Android. Veremos la arquitectura de la plataforma y el modelo de seguridad impuesto por la misma. Luego, observaremos los datos que podemos encontrar en estos dispositivos y las estructuras en que éstos son almacenados. Finalmente, terminaremos mostrando los mecanismos que podemos utilizar para extraerlos.

2.5.1. Arquitectura de la plataforma

La plataforma Android cuenta con una arquitectura dividida en cuatro capas [1], como se muestra en la figura 2.2.

La capa inferior consiste en un kernel Linux modificado, el cual provee una capa de abstracción para interactuar con el hardware. Contiene todos los drivers necesarios para interactuar con los distintos componentes del hardware (cámara, teclado, Wifi, audio, etc).

La segunda capa consta de dos secciones. Por un lado, un conjunto de bibliotecas específicas para cada componente de hardware, las cuales le permiten al dispositivo el manejo de un gran conjunto de datos. Por otro lado, el entorno de ejecución de Android, el cual provee un componente clave para la ejecución de las aplicaciones: *Dalvik Virtual Machine*. Cada aplicación tendrá su propia instancia de Dalvik VM, permitiendo que la aplicación ejecute dentro de un ambiente controlado e independiente. De esta forma, una aplicación no puede interferir indebidamente con otras aplicaciones, con el sistema operativo, ni acceder directamente al hardware del dispositivo.

La tercera capa consta del *Application Framework*, encargado de brindar

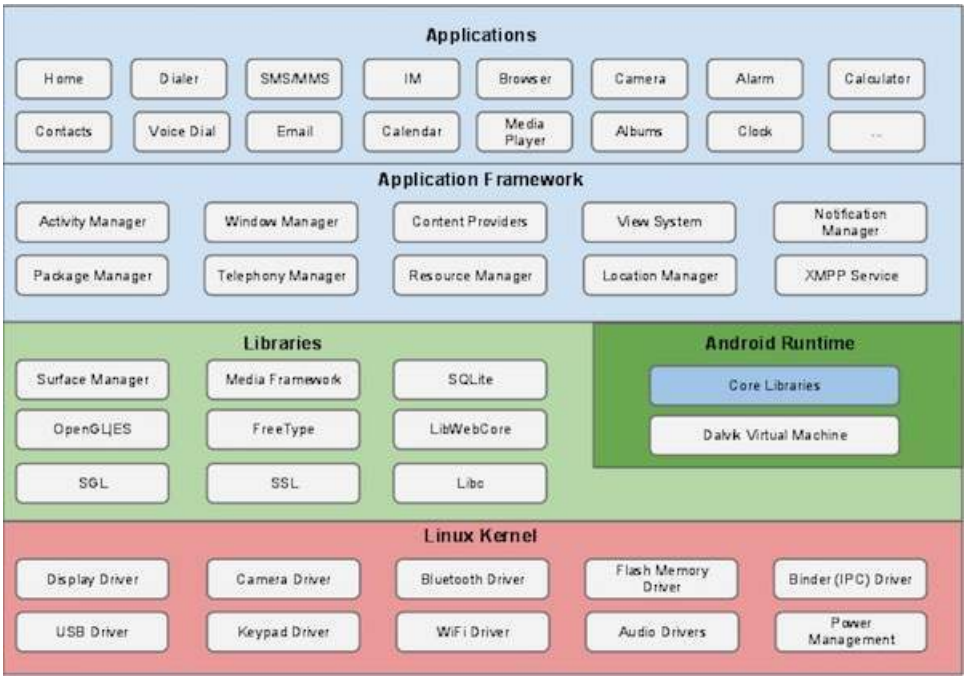

Figura 2.2: Arquitectura de la plataforma Android [7]

muchos de los servicios de alto nivel a las aplicaciones a través de clases java, para que los desarrolladores de aplicaciones puedan usarlos en las mismas. Por ejemplo, uno de estos servicios son los *Content Providers*, los cuales son utilizados por las aplicaciones para comunicar datos a otras aplicaciones.

Finalmente, en la cuarta capa se encuentran las aplicaciones instaladas en el dispositivo. Este conjunto de aplicaciones incluye tanto las pre-instaladas como las instaladas por el usuario.

2.5.2. Información del sistema

Como ya vimos, Android fue creado sobre Linux, por lo que estas dos plataformas cuentan con características en común. Una característica muy útil para obtener datos del sistema, son los logs. Existen varios logs de los cuales se pueden obtener datos del sistema, siendo log *dumpsys* el que brinda más información. Para acceder a este log primero es necesario que el dispositivo se encuentre en modo *debug*. Luego, debemos ejecutar el comando *adb shell dumpsys*.

Entre los datos que se pueden encontrar en este log se destacan cuentas sincronizadas, últimas coordenadas GPS conocidas, notificaciones del dispositivo, información del sistema (versiones del kernel, bootloader, etc), información de telefonía (MEID y ICCID) e información sobre el uso del dispositivo (por ejemplo, últimas aplicaciones utilizadas).

2.5.3. Información de las aplicaciones

Las distintas formas de almacenamiento que pueden ser utilizadas por las aplicaciones son las siguientes [3]:

- **SharedPreferences**: Los SharedPreferences son archivos con formato XML, en los cuales el desarrollador de la aplicación puede almacenar un listado de parejas clave-valor de tipos de datos primitivos (booleans, floats, ints, longs y strings). Suelen ser utilizados para almacenar configuraciones de la aplicación, incluyendo los cambios de configuración realizados por el usuario. Estos datos son mantenidos a través de las sesiones del usuario, incluso cuando se cierra la aplicación.
- **SQLite databases**: Este tipo de bases de datos es el más utilizado por la comunidad Android, debido a que su código es Open Source, compacto y toda la base de datos se encuentra en un único archivo.
- **Otros tipos de archivos**: Es posible que las aplicaciones almacenen otros tipos de archivos (por ejemplo video, audio, texto, etc). Estos archivos pueden almacenarse o bien en memoria interna en un directorio que es privado para la aplicación, o bien en memoria externa (esto es, la SD-Card) en un directorio que es accesible por las demás aplicaciones.

La estructura de directorios con que cuenta una aplicación en su directorio privado en el almacenamiento de memoria interna es la siguiente:

- **databases**: directorio que contiene las bases de datos SQLite manejadas por la aplicación. Además de las bases de datos en sí, se pueden encontrar bases de datos temporales, las cuales son utilizadas por SQLite para realizar rollbacks.
- **cache**: directorio utilizado para almacenar archivos temporales. Estos datos pueden ser borrados automáticamente por la aplicación en caso de que no haya espacio suficiente en la memoria interna.
- **files**: directorio para almacenar archivos creados por la aplicación cuando la misma desea que los mismos no sean públicos para las demás

aplicaciones.

- **shared_prefs**: directorio donde se almacenan los archivos SharedPreferences.

Cabe destacar que las aplicaciones también cuentan con la posibilidad de almacenar datos de forma remota a través de internet. Esto a menudo resulta importante cuando las aplicaciones desean evitar almacenar información sensible en el dispositivo.

2.5.4. Preparación del dispositivo para la extracción: Rooting

En el modelo de seguridad que impone Android, el dispositivo viene de fábrica en un estado en el cual el usuario del mismo no tiene acceso directo a gran parte de los datos que se encuentran en el mismo. De esta forma, para acceder a los datos encontrados dentro del área de almacenamiento privada de las aplicaciones o para crear una disk image del dispositivo, debemos hacer lo que se conoce popularmente como *rooting* del dispositivo.

Rooting es el proceso mediante el cual se obtiene root en un dispositivo Android. El mismo se realiza de la misma forma que otros sistemas Linux: explotando una vulnerabilidad. Tras haber obtenido root, es posible modificar el contenido del dispositivo, acceder a zonas protegidas del mismo ó incluso utilizar recursos de hardware sin las restricciones impuestas por el fabricante.

Existen tres tipos de rooting:

- **Temporal**: el dispositivo vuelve a sus condiciones normales una vez que es reiniciado.
- **Permanente**: se modifican varios archivos del firmware del dispositivo, haciendo que los privilegios se conserven al reiniciar el dispositivo.
- **Recovery mode**: se flashea una ROM customizada de la partición de recuperación, mediante la cual se inicia el sistema con privilegios de root.

Desde el punto de vista de la extracción de datos, no es necesario realizar un rooteo permanente del dispositivo, ya que un rooting temporal brinda el acceso necesario a las áreas protegidas, y los datos modificados por este proceso volverán a su estado original una vez reiniciado el dispositivo.

Algunas de las consideraciones que se deben tener en cuenta cuando se va a realizar el rooting del dispositivo son [39]:

1. Dado que el proceso de rooteo se basa en aprovechar una vulnerabilidad del dispositivo, esta vulnerabilidad depende del modelo y versión de Android del mismo. Por lo tanto, es importante conocer las caracteristicas del mismo antes de realizar el rooteo.
2. Al realizar el rooting del dispositivo se pierden las protecciones del modelo de seguridad de Android, ya que un dispositivo rooteado permite la ejecución de aplicaciones con privilegios totales.
3. Se pueden llegar a modificar áreas del dispositivo en las cuales se guardan datos del usuario.

Sin embargo, a pesar de que existen estos riesgos, el rooting es el proceso que se usa habitualmente si se quieren obtener los datos sensibles encontrados en el dispositivo.

2.5.5. Alternativas a rooting

La primera alternativa es utilizar el comando `adb backup`, el cual no requiere acceso root al dispositivo pero sí que el mismo tenga activada la opción "USB Debugging" en su menú de configuración. De esta forma, este comando nos permite realizar un respaldo de los datos internos de las aplicaciones o de la tarjeta SD del dispositivo, y también se puede indicar que se extraiga el apk de las mismas.

Para completar el respaldo con éxito, se requiere intervención por parte del usuario del dispositivo móvil. Al momento de ejecutar el comando desde la consola de la PC, en la pantalla del dispositivo se muestra un diálogo, donde se solicita confirmación.

Los desarrolladores de aplicaciones cuentan con la posibilidad de deshabilitar el backup de datos de su aplicación, esto es, no permitir el respaldo de los datos que la aplicación mantiene en su región de almacenamiento interno privado. Sin embargo, observamos que la mayoría de las aplicaciones suelen tener la opción de respaldo habilitada.

La segunda alternativa es instalar una *CRMI* (Custom Recovery Mode Image). El modo de recuperación de Android fue diseñado para realizar actualizaciones, formatear y realizar operaciones de mantenimiento sobre el disposi-

tivo. Las funcionalidades brindadas por este modo son limitadas, y ninguna de ellas brinda la capacidad de obtener privilegios de root. Sin embargo, mediante la instalación de particiones customizadas de recuperación, se pueden obtener estos privilegios. Una vez que se tiene una CRMI, se la flashea sobre la partición de booteo, y se reinicia el dispositivo. Luego de iniciado, se cuenta con privilegios de root, pudiéndose obtener todos los datos deseados. Dado que no es necesario montar la partición de datos del usuario, la copia puede realizarse sin correr el riesgo de que se alteren los datos, incluso datos borrados. Luego de realizar la extracción deseada, se debe volver a flashear la versión original de la partición de booteo [31].

Capítulo 3

Análisis del problema

3.1. Introducción

Como vimos en el capítulo anterior, el área de *mobile forensics* no cuenta con procedimientos tan bien definidos a la hora de realizar una extracción de datos como se cuenta en el caso de los PC.

Por la naturaleza del ecosistema Android, los dispositivos móviles son diseñados por una gran diversidad de fabricantes. Los mismos poseen cierto grado de libertad a la hora de implementar las funcionalidades que ofrecen a través del sistema operativo Android. Esto lleva a que frecuentemente nos encontremos con componentes que son implementados de diferente forma por distintos fabricantes. Por ejemplo, un fabricante puede optar por utilizar un filesystem distinto al de referencia (EXT4) para la partición que almacena los datos del usuario [5]. Estas diferencias, que para el usuario final del dispositivo no suelen ser perceptibles, tienen un impacto considerable en la forma de extraer los datos.

Vemos entonces que a pesar de que los tipos de datos que podemos extraer de los dispositivos Android son en esencia los mismos, la forma en la cual debemos realizar su extracción muy a menudo varía dependiendo de variables como son el fabricante del dispositivo y la versión del sistema operativo Android.

La dependencia en la implementación por parte del fabricante se hace mayor si consideramos realizar extracciones físicas. En el caso de extracciones lógi-

cas, tenemos una menor dependencia con la implementación del fabricante por el simple hecho de contar con las abstracciones y convenciones que establece la plataforma Android, pero aún así existen muchos aspectos en los cuales seguimos dependiendo de la implementación del fabricante también a este nivel. A modo de ejemplo, un caso habitual es el esquema de la base de datos utilizada para almacenar los mensajes de texto. Dicho esquema suele variar dependiendo el fabricante e incluso entre carriers [23].

Por otro lado, también dependemos de la versión de Android que posee el dispositivo. A menudo se producen cambios en la plataforma que provocan que debamos modificar la forma en la que extraemos los datos ya sea porque cambia la forma en que se acceden o presentan los datos. Por ejemplo, en la versión 4.1 de Android se restingió el acceso a logs de aplicaciones. Previo a esto, aplicaciones de terceros tenían la capacidad de solicitar el permiso *READ_LOGS* con el cual podían acceder a los logs generados por las demás aplicaciones [15].

Por lo tanto, vemos que contar con un nivel de abstracción que permita concentrarnos en qué datos deseamos extraer, sin tener que estar considerando cómo extraerlos de cada dispositivo en particular, sería de gran utilidad para poder desarrollar herramientas y procesos forenses que escalen ante la gran variedad del ecosistema Android.

Asimismo, tener este nivel de abstracción no sólo resultaría útil para uniformizar el proceso de extracción de datos a través de diversos dispositivos y versiones de Android, sino que también podría facilitar la forma en que especificamos los datos que deseamos extraer. De esta forma, sería interesante poder realizar extracciones especificando únicamente el tipo y la fuente de los datos a obtener.

Por último, vemos que este nivel de abstracción también nos brinda la oportunidad de poder "empaquetar" las técnicas de extracción de datos de una forma uniforme. Esto permite que luego de desarrolladas puedan ser compartidas y reutilizadas por otros. Este aspecto resulta vital para crear un ambiente de desarrollo colaborativo el cual permita soportar una porción considerable del ecosistema Android.

Las consideraciones arriba descritas respaldan nuestra convicción de la necesidad de contar con una herramienta que asista en la extracción de datos de dispositivos móviles Android podría favorecer sustancialmente el trabajo del investigador forense. En la siguiente sección describiremos los problemas que

apuntamos resolver mediante la creación de la misma.

3.2. Objetivos planteados

El objetivo principal de la herramienta propuesta es **facilitar la extracción de datos** de dispositivos móviles Android. Para esto, la herramienta deberá realizar una fuerte separación entre cómo el usuario expresa lo que desea extraer y cómo se lleva a cabo la extracción. Esta separación permite a los usuarios concentrarse en qué datos extraer y no en cómo debe implementar el procedimiento de extracción. De esta forma, evitamos que los usuarios requieran contar con amplio conocimiento técnico para utilizar la herramienta, lo cual permite que la misma sea utilizada por un espectro más amplio de investigadores forenses.

Como vimos, una de las características fundamentales del ecosistema Android es su diversidad. Esto hace que sea importante abordar el desafío de extraer datos de forma escalable desde la concepción de la herramienta. Para esto, la misma deberá contar con una arquitectura que **facilite su extensibilidad** con el fin de poder soportar rápidamente nuevas formas de extraer datos. Esta es una necesidad clave ante la permanente evolución de Android y su ecosistema de dispositivos.

Finalmente, otro aspecto clave a resolver es el hecho de que hoy en día las herramientas forenses cuentan con escasa interoperabilidad entre ellas. Independientemente de los motivos comerciales o políticos que haya detrás de este hecho, consideramos que uno de los aspectos más importantes para que la herramienta resulte útil en la práctica radica en **facilitar la manipulación de los datos extraídos**. Por lo tanto, la herramienta debe brindar sus salidas en un formato que facilite la consumición de las mismas por parte de herramientas de análisis forense. Por esta razón, creemos razonable buscar la adopción de lenguajes o formatos estándares en el área para expresar las salidas de la herramienta.

3.3. Enfoque inicial

Para cumplir con los objetivos propuestos nos planteamos cómo debería ser la interacción que proporciona la herramienta y las operaciones que la misma brinda. A continuación analizaremos cómo llegamos a decidir varios de los aspectos claves que luego veremos reflejados en el funcionamiento de la

herramienta.

Si consideramos el comportamiento más simple que uno esperaría de una herramienta de este tipo, seguramente sería el de contar con un conjunto de operaciones de extracción definidas y ser capaz de extraer exactamente los datos que deseamos con una única operación. Esto nos lleva a considerar cómo dividiríamos el dominio de operaciones de extracción con el que debemos contar. Al considerar las variables descritas anteriormente, surgen naturalmente dos enfoques que coinciden con casos de uso habituales para un investigador:

- Por un lado, nos puede interesar realizar extracciones según la **fuente de datos**. Por ejemplo, un investigador podría desear obtener todos los datos relevantes de la aplicación Facebook.
- Por otro lado, también puede ser interesante poder realizar extracciones según su **tipo de datos**. Por ejemplo, un investigador podría desear obtener todas las imágenes del dispositivo (ya sean estas de Facebook, WhatsApp, la cámara, etc).

En vez de tener que decidir por uno de los dos enfoques, vimos que sería muy interesante intentar satisfacer ambas necesidades. Para esto, nos dimos cuenta que esto es posible si consideramos operaciones de extracción concisas que luego puedan ser combinadas para obtener los datos deseados. En vez de intentar dividir el conjunto de operaciones utilizando únicamente una de estas variables (esto es, en operaciones dedicadas a extraer datos o bien según su tipo de dato, o bien según su fuente de datos), lo dividiríamos según ambas.

Finalmente, vemos que de esta forma la herramienta contaría con operaciones más granulares, pero al combinarlas nos permitirían extraer con precisión los datos deseados. Por lo tanto, logramos así que la herramienta tenga una mayor flexibilidad al abarcar ambos casos de uso. De hecho, este enfoque permite que la misma tenga una mayor capacidad de expresión, pudiendo llegar a ser bien específicos en los datos que deseamos extraer al fijar tanto tipo de dato como fuente de datos. Por ejemplo, nos permitiría expresar tanto el deseo de extraer únicamente las imágenes de Facebook, como el de obtener todos los mensajes de texto del dispositivo (sin importar si estos provienen de Hangouts ó WhatsApp).

En resumen, el enfoque que tomamos para concebir una operación nos ofrece lo siguiente:

- **Expresividad**: Permite tener un nivel de precisión muy interesante en los datos que deseamos extraer.
- **Flexibilidad**: Permite cubrir distintos casos de uso de forma simple.
- **Extensibilidad**: Posibilita un diseño que permite ser extendido para soportar nuevas propiedades de operaciones.

3.4. Conceptos fundamentales

Empecemos por detallar mejor dos de los conceptos claves que mencionamos. Estos son: fuente de dato y tipo de dato. En base a los mismos es que determinamos la granularidad que tendrán las operaciones, como vimos en el enfoque recién planteado.

La información que solemos encontrar en los dispositivos móviles puede provenir de diversas **fuentes de datos**. A su vez, resulta interesante observar que hay conjuntos de fuentes de datos para los cuales el mecanismo para obtener datos de las mismas es el mismo. En estos casos, decimos que dichas fuentes son del mismo **tipo de fuente de dato**.

Por otro lado, las diversas piezas de información que podemos encontrar en los dispositivos poseen una semántica diferente para el usuario dependiendo del concepto que representan. Las piezas de información que representan un mismo concepto las asociamos a un **tipo de dato**.

En el enfoque planteado, vimos que cada operación realiza un trabajo conciso. De esta forma, una **operación de extracción** extrae un único tipo de dato, de una única fuente de datos y para determinado rango de versiones y modelos de dispositivos Android.

Ahora que hemos visto lo que hace una operación de extracción, resulta importante que precisemos el término utilizado para referirnos a ella. Si bien la herramienta fue enfocada desde su concepción a la extracción de datos, esta no es exclusivamente la única tarea que realiza una operación.

Además de extraer los datos originales, deseamos encontrar únicamente los de cierto tipo de dato y luego representar la información encontrada sobre los mismos en cierto lenguaje. Estas tareas no caen exclusivamente en la etapa que se conoce como extracción en la mayoría de los modelos de proceso forense. Tomemos para esto como referencia el modelo propuesto y utilizado por el Departamento de Justicia de EEUU [40] y veamos las tres etapas del

proceso forense en que participa una operación:

- **Extracción**, dado que se encarga de obtener datos de una fuente de datos del dispositivo.
- **Examinación**, dado que identifica y presenta datos hallados (correspondientes con el tipo de dato de la operación), en un nivel de abstracción mayor al de los datos obtenidos originalmente.
- **Presentación**, dado que los datos hallados serán expresados en cierto lenguaje estándar.

3.5. Requerimientos

A continuación describiremos los requerimientos con los cuales consideramos que debe cumplir la herramienta para lograr los objetivos planteados.

A. Funcionalidades claves

La herramienta debe proveer al usuario un conjunto de operaciones que faciliten la extracción de datos de dispositivos móviles Android. Para esto, la misma debe brindar las siguientes funcionalidades claves:

A1. Operaciones atómicas

Las operaciones deben contar con un alto nivel de granularidad. De esta forma, una operación debe extraer un único tipo de dato, de una fuente de datos determinada, soportando un determinado conjunto de dispositivos y versiones de Android.

A2. Consulta de operaciones

La herramienta debe contar con la posibilidad de consultar (y filtrar) por las operaciones disponibles. De esta forma, el usuario puede descubrir las mismas y sus características.

A3. Ejecución de operaciones

La herramienta debe permitir ejecutar una serie de operaciones en un único paso. Esto permite al usuario combinar operaciones y, debido al alto nivel de granularidad que poseen las mismas, tener la capacidad de extraer con gran precisión los datos deseados.

A4. Modos de ejecución de la herramienta

La herramienta debe contar con dos modos de ejecución:

- Un **modo interactivo**, en el cual el usuario pueda realizar consultas sobre las operaciones disponibles filtrando por los parámetros de las mismas y luego ejecutar las operaciones deseadas.
- Un **modo batch**, en el cual el usuario ya conoce las operaciones que desea utilizar y únicamente especifica cuáles ejecutar.

B. Extensibilidad

La herramienta debe proveer mecanismos que permitan extender fácilmente los siguientes tres puntos:

1. Tipos de datos
2. Tipos de fuente de datos
3. Operaciones

C. Acceso de datos

La herramienta debe ser capaz de abstraer la forma en que accede a los datos. De esta forma, posibilita su interoperación tanto con la diversidad de dispositivos reales que hay como con emuladores, y además permite la extensibilidad de la herramienta para soportar otros dominios de datos.

D. Salida de datos

D1. Lenguaje estándar

La herramienta debe hacer uso de un lenguaje estándar para expresar sus salidas de datos, con el fin de que para otras herramientas sea sencillo consumir los datos producidos.

D2. Lenguaje extensible

Además, el lenguaje utilizado para representar los datos de salida debe ser extensible, dado que es necesario que el mismo sea capaz de representar información de nuevos tipos de datos.

3.6. Casos de uso

Vistos los requerimientos, ahora vamos a considerar los casos de uso necesarios para satisfacer a los mismos.

Comencemos por describir al usuario el cual apuntamos que haga uso de nuestra herramienta. El mismo lo concebimos como un investigador forense, el cual no necesariamente cuenta con conocimiento técnico de cómo se implementa cada operación.

Deseamos que este usuario pueda hacer uso de la herramienta principalmente de dos posibles formas:

1. Manual, esto es, interactuando directamente con la herramienta con el fin de obtener los datos que desea de un cierto dispositivo.
2. Automática, con el fin de permitirle automatizar tareas de obtención de datos o facilitar la integración de la herramienta con otras herramientas utilizadas por el usuario.

Para estos dos casos de uso es que la herramienta posee dos modos distintos de ejecución, interactivo y batch respectivamente, que son establecidos en (Req. A4).

En tanto, en (Req. B) se expresa la capacidad de extensibilidad que debe tener la herramienta. Veremos cómo el sistema tiene dicha capacidad mediante dos casos de uso que permiten agregar y eliminar operaciones del sistema. Estos se

centran en el punto de extensión que es de mayor interés para la herramienta. También consideraremos los otros dos puntos de extensión, que auspician de soporte al anterior, en el caso de agregar una operación y que la misma haga uso de un tipo de dato o tipo de fuente de dato nueva al sistema.

Finalmente, el usuario tiene a su disposición el siguiente conjunto de comandos para interactuar con el sistema:

Comando	Descripción
set_device_info	*Permite especificar la información del dispositivo en el modo de ejecución interactivo. Esto hace posible que podemos omitir dicha información de los comandos* `list` *y* `execute` *posteriores.*
list	*Lista las operaciones disponibles del sistema. Es de especial valor su uso en el modo interactivo, ya que permite al usuario descubrir las operaciones y sus características.*
execute	*Permite ejecutar un conjunto de operaciones tanto en modo interactivo como en modo batch.*
add_ext	*Permite añadir una nueva extensión al sistema, ya sea operación, tipo de dato o tipo de fuente de datos.*
rm_ext	*Permite eliminar una extensión del sistema. En el caso de tipo de datos y tipo de fuentes de datos, la eliminación de la extensión se llevará a cabo únicamente en caso que la misma no esté siendo utilizada por ninguna operación. Si deseamos eliminarla, primero será necesario eliminar explícitamente las operaciones que hacen uso de la misma.*

Cuadro 3.1: Comandos disponibles de la herramienta

3.6.1. Ejecución de operaciones en modo interactivo

Flujo principal

1. El usuario ejecuta la herramienta sin ningún parámetro adicional.
2. El sistema inicia una sesión de ejecución interactiva.
3. El usuario ejecuta el comando `list` indicando el tipo de datos, la fuente de datos (y la información del dispositivo en caso de no haber utilizado el comando `set_device_info`).

4. El sistema retorna la información de todas las operaciones que cumplen con los valores indicados. Para cada operación se muestra:
   - Nombre de la operación.
   - Tipo de datos que examina.
   - Fuente de datos utilizada para extraer los datos.
   - Modelos de dispositivos soportados
   - Versiones de Android soportadas.
5. El usuario ejecuta el comando `execute` indicando los nombres de las operaciones que desea ejecutar (y la información del dispositivo en caso de no haber utilizado el comando `set_device_info`).
6. Para cada operación ejecutada:
   - Si completó exitosamente, el sistema indica la ruta al directorio en donde fue almacenada la información obtenida.
   - Sí falló, el sistema indica la causa.
7. El usuario no desea realizar más extracciones.
8. Fin del caso de uso.

Flujos alternativos

- 3A.
  1. El usuario ejecuta el comando `set_device_info` indicando la información del dispositivo.
  2. El sistema guarda la información del dispositivo. Esta información será tenida en cuenta por los comandos `list` y `execute`.
  3. Retorna al paso 3 del flujo principal.
- 5A.
  1. El usuario ejecuta el comando `execute` indicando los nombres de las operaciones que desea utilizar, pero no indica la información sobre el dispositivo.
  2. El sistema retorna un mensaje de error indicando que es necesario conocer la información del dispositivo.
  3. Se retorna al paso 5 del flujo principal.
- 7A. El usuario desea realizar otra extracción.
  1. Se retorna al paso 3 del flujo principal.

3.6.2. Ejecución de operaciones en modo batch

La diferencia más importante entre este modo de ejecución de operaciones y el modo previamente explicado consiste en que en este modo el usuario ya sabe cuales son las operaciones que desea ejecutar. Por lo tanto, no necesita de un modo interactivo en donde realizar consultas previo a ejecutar las operaciones.

Flujo principal

1. El usuario ejecuta la herramienta pasándole como parámetro el comando `execute` seguido de los nombres de las operaciones que desea ejecutar y la información sobre el dispositivo.
2. Para cada operación ejecutada:
   - Si completó exitosamente, el sistema indica la ruta al directorio en donde fue almacenada la información obtenida.
   - Sí falló, el sistema indica la causa.
3. Fin del caso de uso.

Flujos alternativos

- 1A.
  1. El usuario ejecuta la herramienta pasándole como parámetro `execute` sin indicar los nombres de las operaciones a ejecutar y/o la información sobre el dispositivo.
  2. El sistema retorna un mensaje de error indicando los datos omitidos.
  3. Fin del caso de uso.

3.6.3. Adición de una extensión

Flujo principal

1. El usuario ejecuta (ya sea en modo batch ó interactivo) el comando `add_ext` pasándole como parámetros el tipo de extensión (operación, tipo de dato o tipo de fuente de datos) que desea agregar y la ruta a la definición de la nueva extensión.

2. El sistema valida la definición de la nueva extensión, la agrega y le indica al usuario que la misma fue agregada exitosamente.

3. Fin del caso de uso.

Flujos alternativos

- 1A.
  1. El usuario desea agregar una operación y la misma utiliza un tipo de dato o un tipo de fuente de datos que no existe en el sistema.
  2. El sistema le indica al usuario que la definición de la operación hace uso de una extensión que no se encuentra en el sistema y la misma debe ser agregada previamente para poder añadir esta operación.
  3. Fin del caso de uso.
- 2A. / 1A.2A
  1. El sistema determina que la definición de la extensión recibida es inválida.
  2. El sistema retorna un mensaje de error indicando los campos inválidos de la definición.
  3. Fin del caso de uso.

3.6.4. Eliminación de una extensión

Flujo principal

1. El usuario ejecuta (ya sea en modo batch ó interactivo) el comando `rm_ext` pasándole como parámetros el tipo de la extensión que desea eliminar y su nombre.

2. El sistema elimina la extensión e informa al usuario de ello.

3. Fin del caso de uso.

Flujo alternativo

- 1A.
  1. El usuario ejecuta el comando `rm_ext` sin indicar el tipo y/o nombre de la extensión que desea eliminar.

2. El sistema retorna un mensaje de error haciendo referencia a los parámetros faltantes.
3. Fin del caso de uso.

3.6.5. Diagrama de casos de uso

De forma de resumir la interacción entre el usuario y el sistema, en la figura 3.1 se muestran los casos de uso en base a dos roles:

- El primero de ellos trata del rol *usuario*, el cual permite utilizar al sistema tanto en modo interactivo como en modo batch.
- El segundo trata del rol *administrador*, el cual permite configurar la herramienta agregando y removiendo extensiones de la misma.

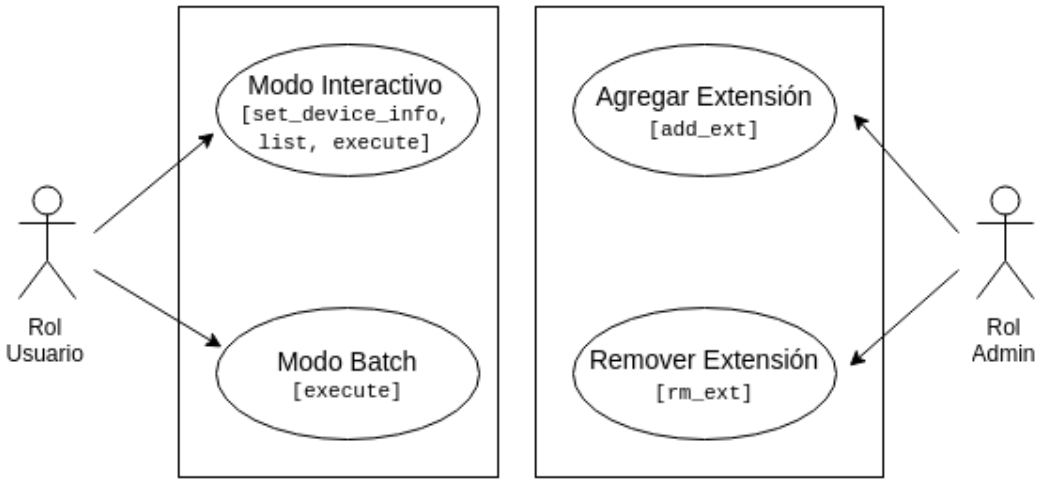

Figura 3.1: Diagrama de casos de uso

3.7. Definición del alcance

En base a los requerimientos establecidos, veremos a continuación varios aspectos que merecen ser mencionados con el fin de ayudar a delimitar con claridad el alcance del proyecto.

A. Funcionalidades claves

Como vimos en el requerimiento A1, la herramienta contará con un conjunto de operaciones las cuales realizan la extracción, examinación y presentación de los datos encontrados. A través de dichas operaciones, la herramienta manejará:

- **Datos extraídos de dispositivos móviles** de carácter lógico y orientados a aquellos producidos por los usuarios de los mismos (esto es, no nos enfocaremos en examinar datos producidos por malware, por ejemplo). De todas formas, la arquitectura de la herramienta será diseñada de manera de facilitar la incorporación de otros dominios de datos.
- **Datos sobre el proceso de extracción**. En primer lugar, será de interés conocer la relación entre los datos que fueron originalmente extraídos del dispositivo y la información examinada (esto es, dada una pieza de información examinada deseamos conocer de qué fuente, por ejemplo de qué archivo, se extrajo dicha información). En segundo lugar, será de interés contar con información sobre quién, cuándo, cómo y dónde se utilizó la herramienta para extraer los datos. Para esto, buscaremos incluir información sobre los colaboradores, fecha y hora de la extracción y datos del ambiente utilizado.

En cuanto al mecanismo de consulta de operaciones descrito en A2, el prototipo de la herramienta contará con capacidades básicas de filtrado. Seguramente pueda ser útil contar con opciones más avanzadas de filtrado (como expresiones lógicas) con el fin de facilitar el uso de la herramienta, pero consideramos que esto sería más adecuado hacerlo en una segunda iteración.

B. Extensibilidad

Tipos de fuentes de datos

El prototipo de la herramienta estará enfocado en la extracción de datos de aplicaciones. Para esto, contaremos con un tipo de fuente de datos que llamaremos *Application* y el cual permitirá tomar como fuente de datos aquellos datos pertenecientes a una aplicación instalada en el dispositivo.

Cabe destacar que, de todas formas, será fundamental que la herramienta tenga un diseño extensible que le permita soportar otros tipos de fuentes de datos como establecimos en (Req. B). Por esta razón, vemos que como contra-

partida a datos de aplicaciones, sería interesante desarrollar (en una futura iteración de la herramienta) la extracción de datos del sistema operativo. Esta posibilidad se expresa en más detalle en la sección 7.5.

Tipos de datos

El prototipo de la herramienta por defecto contará con un conjunto básico de tipos de datos correspondientes a los diversos objetos predefinidos por el lenguaje.

A modo de ejemplo, veamos a continuación el estilo de tipos de datos que pretenderemos poder representar mediante el mismo, ya sea utilizando objetos predefinidos o extendiendo el lenguaje.

Para el dominio de *datos de aplicaciones*, desearíamos poder representar:

- Mensajes de texto (SMS, MMS)
- Mensajes de chat (Hangouts, WhatsApp, Facebook)
- Mensajes de correo (Android default email app, GMail)
- Contactos (Android default agenda app, Facebook, GMail)
- Registro de llamadas
- Eventos de calendario (Google Calendar)
- Archivos (Locales, GDrive, Dropbox)
- Información de ubicación (GPS, geo-fencing)
- Historial de navegación (Chrome, Firefox)

Para el dominio de *datos del sistema*, desearíamos poder representar:

- Datos de telefonía (ICCID, IMEI, MEID)
- Información de red (Dirección MAC de Wi Fi, Bluetooth)
- Información del dispositivo (Modelo, marca)
- Procesos corriendo
- Conexiones activas

En el primer grupo vemos ejemplos de tipos de datos que podemos obtener de fuentes de datos del tipo de *Application* como con los que nos enfocaremos. En el segundo, vemos ejemplos de datos relevantes al sistema, que si

bien no los consideraremos directamente, ya mencionamos que es importante que el lenguaje pueda representarlos para que luego sea posible extender la herramienta y soportar su uso.

Operaciones

El prototipo de la herramienta contará con un conjunto de operaciones por defecto, las cuales serán desarrolladas como parte del caso de estudio.

C. Acceso de datos

Como vimos, este requerimiento busca abstraer la forma utilizada por la herramienta para acceder a los datos con el fin de poder interoperar con múltiples tipos de dispositivos (incluso con emuladores).

En este sentido, el protocolo ADB (Android Debug Bridge) y sus herramientas nos proveen la abstracción necesaria que precisamos para cumplir con el requerimiento (Req. C). El mismo nos posibilita el intercambio de datos y ejecución de comandos de forma transparente, tanto en dispositivos físicos como en emuladores.

Para el ambiente de desarrollo del prototipo, utilizaremos un emulador Android por la flexibilidad, facilidad y control que esto nos brinda. En particular:

- Nos da la posibilidad de crear imágenes específicas del estado de un dispositivo.
- Evitamos preocuparnos por fallas o pérdida de datos de un dispositivo real.
- Podemos contar con acceso root al dispositivo de manera muy sencilla, lo cual nos permitirá acceder a todos los datos del mismo.

De todas formas, la herramienta será diseñada para soportar tanto dispositivos reales como virtuales (emuladores). El hecho de contar con una interfaz de acceso a los dispositivos como es *adb* y un emulador que imita al hardware de un dispositivo real hace que esta transición, en teoría, no deba requerir esfuerzo adicional.

D. Salida de datos

Debido al funcionamiento que comprende una operación, vemos que la herramienta deberá contar con los siguientes tipos de salida de datos:

- **Datos extraídos**, para expresar los datos que fueron obtenidos del dispositivo en su forma original.
- **Datos examinados**, para expresar los datos encontrados a partir de los anteriores.
- **Datos del proceso de extracción**, para expresar información acerca de los colaboradores que obtuvieron los datos examinados, el ambiente que utilizaron y los archivos de los cuales fue obtenida la información de los mismos.

De manera de cumplir con los requerimientos [Req. D1] y [Req. D2], utilizaremos el lenguaje CybOX para representar a las salidas de datos examinados y del proceso. El lenguaje ya fue analizado extensamente en la sección 2.4.1 en donde vimos cómo el mismo cumple con los requerimientos mencionados (esto es, es estándar y extensible). Además, vimos como el lenguaje posibilita la representación de los tipos de datos mencionados en los puntos A y B de esta sección de Definición del Alcance.

Capítulo 4

Diseño de la herramienta

4.1. Introducción

Este capítulo describe el diseño que elaboramos para la herramienta propuesta, la cual denominaremos *Android Inspector*. El mismo fue elaborado a partir del análisis realizado en el capítulo anterior.

Comenzaremos por presentar el modelo de datos con que cuenta la herramienta. La mayoría de éstos representan una traducción casi directa de los conceptos vistos en la sección 3.4. Posteriormente, describiremos la arquitectura de la herramienta profundizando en las responsabilidades de cada componente y sus interfaces. Luego, veremos cómo se realiza la persistencia de los diversos recursos que maneja la herramienta. Finalmente, describiremos los diversos puntos de extensión que brinda la herramienta.

Antes de comenzar con el modelo de datos y a modo de fijar conceptos, veamos una vista general del funcionamiento de la herramienta realizando la tarea principal, esto es, ejecutando una operación.

En la figura 4.1 podemos observar las interacciones que se producen al ejecutar una operación. Viendo el sistema desde afuera, el usuario ingresa el nombre de la operación que desea ejecutar junto a la información del dispositivo y el sistema la ejecuta. En caso de éxito, esta ejecución produce un directorio con los datos extraídos en crudo y los archivos XML conteniendo los datos examinados expresados en CybOX.

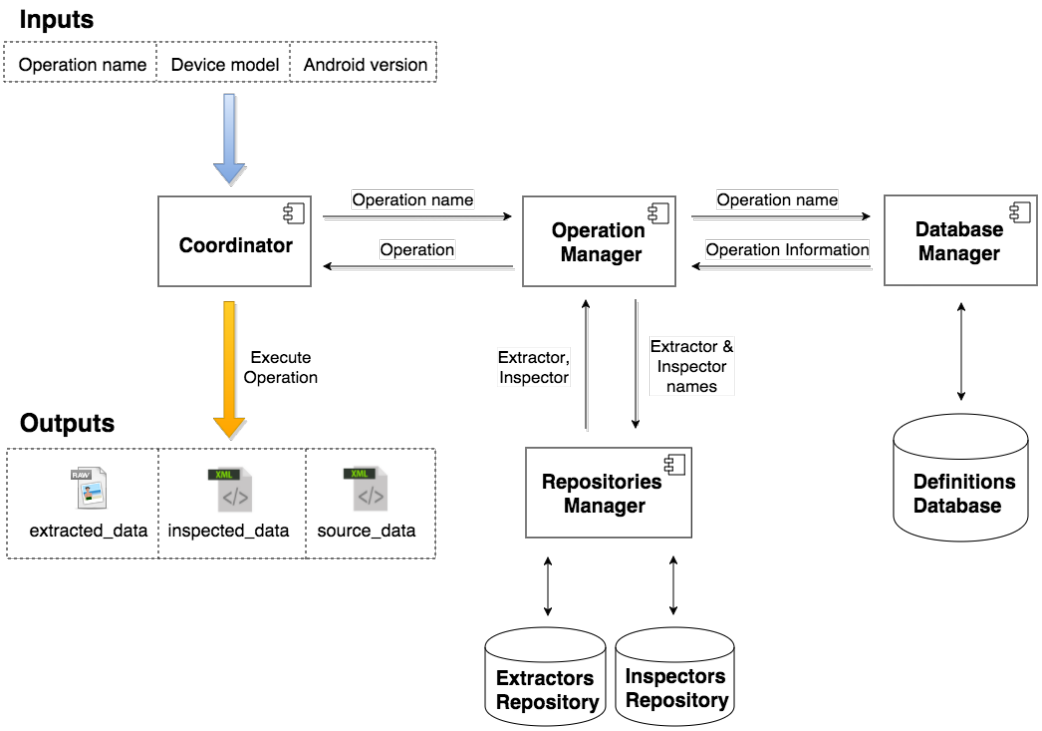

Figura 4.1: Funcionamiento de alto nivel al ejecutar una operación

Observemos lo que sucede internamente:

- El componente *Coordinator* obtiene los datos de la capa de presentación. Utiliza el nombre de la operación para pedirle al *OperationsManager* que le construya una instancia de la misma. Luego, la ejecuta pasándole la información del dispositivo, produciendo los datos de salida que se pueden observar en la figura, e informa al usuario del resultado de la operación.
- El *OperationsManager* a su vez, se encarga de construir una operación. Para eso, debe recurrir al *DatabaseManager* para obtener la información de la misma. Luego, con esta información le solicita al *RepositoriesManager* las instancias de los componentes necesarios para construir dicha operación (extractor e inspector).
- Los componentes *DatabaseManager* y *RepositoriesManager* realizan las tareas de persistir y recuperar datos. El *DatabaseManager* se encarga de las definiciones de las extensiones mientras que el *RepositoriesManager* de las extensiones en sí.

4.2. Modelo de datos

Comenzaremos por describir las estructuras de datos utilizadas por los componentes del sistema.

4.2.1. DataSource

Estructura utilizada para facilitar la representación de una fuente de dato, esto es, su tipo y valores correspondientes a los parámetros requeridos por el tipo de la misma.

Campo	Tipo de dato	Descripción
type	*string*	*Especifica el tipo de la fuente de la cual se extraen los datos.*
info	*dict(string, string)*	*Especifica los valores de los parámetros de la fuente de datos.*

Cuadro 4.1: Estructura DataSource

De esta forma, una posible instancia de fuente de dato podría contar con: type = 'Application' y parameters = {('package_name', 'com.example.app')}.

4.2.2. OperationInfo

Estructura utilizada para representar la información de una operación. La misma es utilizada por el comando `list` al desplegar dicha información al usuario.

Campo	Tipo de dato	Descripción
name	*string*	*Especifica el nombre de la operación.*
data_type	*string*	*Especifica el tipo de datos que extrae.*
data_source	*DataSource*	*Especifica la fuente de datos de la cual extrae los datos.*
supported_device_models	*list(string)*	*Especifica el conjunto de dispositivos que soporta.*

supported_os_versions	*list(string)*	*Especifica el conjunto de versiones del sistema operativo Android que soporta.*

Cuadro 4.2: Estructura OperationInfo

4.2.3. DeviceInfo

Estructura utilizada para facilitar la representación de la información de un dispositivo. La misma es utilizada en una gran diversidad de comandos.

Campo	Tipo de dato	Descripción
os_version	*string*	*Especifica la versión del sistema operativo Android siendo utilizada por el dispositivo.*
device_model	*string*	*Especifica el modelo del dispositivo.*

Cuadro 4.3: Estructura DeviceInfo

4.2.4. ObjectProperties

Estructura abstracta provista por CyBOX. Sus diversas implementaciones permiten representar el conjunto de propiedades con el que cuenta cada tipo de objeto CybOX. Nosotros hacemos uso de los objetos CybOX como forma de representar la información de cada tipo de dato.

Veremos que CybOX cuenta con un conjunto de objetos (predefinidos en su estándar) por defecto. La herramienta inicialmente importa dichos objetos y genera sus correspondientes tipos de datos, de forma que los mismos queden disponibles en el sistema.

Además de los objetos predefinidos, CybOX permite definir nuevos tipos de objetos. El uso de los mismos será necesario para representar nuevos tipos de datos que deseemos definir.

4.2.5. Object

Estructura provista por CybOX. La utilizamos para representar cada pieza de información de un cierto tipo de dato encontrada como resultado de una examinación.

Campo	Tipo de dato	Descripción
id	*UUID*	*Especifica un identificador único para el objeto.*
properties	*ObjectProperties*	*Especifica el conjunto de propiedades del objeto (las cuales dependen de su tipo).*
related_objects	*list(RelatedObject)*	*Si bien permite declarar varias relaciones con otros objetos, nosotros lo utilizamos únicamente para especificar el archivo del cual fue extraída la información presente en este objeto.*

Cuadro 4.4: Estructura Object

Vistas las estructuras de datos, a continuación detallamos las clases fundamentales utilizadas para llevar a cabo la ejecución de una operación.

4.2.6. Extractor

Interfaz que debe ser implementada por los extractors de las operaciones. Cada extractor implementa el mecanismo de extracción de datos para un determinado tipo de fuente de datos.

Método	Entrada	Salida
execute	*string, dict(string, string)*	*None*

Cuadro 4.5: Interfaz Extractor

El método `execute` es utilizado por la clase *Operation* para extraer los datos del dispositivo. Cuando la clase *Operation* llama a este método, le indica por parámetro la ruta en donde se deben almacenar los datos extraídos y un diccionario conteniendo los valores de cada uno de los parámetros requeridos

por el tipo de fuente correspondiente del extractor.

4.2.7. Inspector

Interfaz que debe ser implementada por el inspector de una operación. Cada inspector implementa el mecanismo a través del cual una operación examina datos provenientes de una determinada fuente de datos y en busca de un determinado tipo de dato.

Método	Entrada	Salida
execute	*DeviceInfo, string*	*list(Object), list(FileObject)*

Cuadro 4.6: Interfaz Inspector

El método `execute` de la interfaz es utilizado por la clase *Operation* para realizar la examinación de los datos extraídos. De esta forma, dicha clase llama al método execute indicando por parámetro la información del dispositivo y la ruta al directorio donde fueron almacenados los datos extraídos. Como salida el método devuelve:

- Una lista de objetos CybOX del tipo correspondiente al tipo de dato de la operación con la información encontrada.
- Una lista de *FileObject* que especifica los archivos de los cuales fue obtenida la información presente en los objetos de la lista anterior.

4.2.8. Operation

Esta clase es utilizada para representar una operación a ejecutar. La misma contiene las instancias de su correspondientes extractor e inspector, así como los valores de los parámetros requeridos por el extractor. También es la encargada de coordinar la ejecución y almacenamiento de los datos producidos por la operación que representa.

Campo	Tipo de dato	Descripción
extractor	*Extractor*	*Extractor concreto a ser utilizado por la operación.*

inspector	*Inspector*	*Inspector concreto a ser utilizado por la operación.*
param_values	*dict(string, string)*	*Diccionario que contiene los valores de esta operación para los parámetros del tipo de fuente de dato. Los mismos son utilizados por el extractor.*

Cuadro 4.7: Clase Operation

Método	Entrada	Salida
execute	*DeviceInfo, string*	*None*

Cuadro 4.8: Métodos de la clase Operation

El método execute recibe como parámetro la información sobre el dispositivo y la ruta al directorio en donde se deben almacenar los datos obtenidos. De esta forma:

1. Ejecuta el método execute del extractor, indicándole el subdirectorio en donde debe almacenar los datos extraídos.
2. Posteriormente, ejecuta el método execute del inspector, indicándole el subdirectorio en donde fueron almacenados los datos extraídos previamente.
3. Luego, almacena los datos producidos por el inspector en los siguientes dos archivos dentro del directorio que le fue indicado:
   - *inspected_ data.xml*, conteniendo los objetos que representan los datos encontrados en la examinación.
   - *source_ data.xml*, conteniendo los objetos que representan los archivos de los cuales contienen la información presente en los examinados.

Por lo tanto, el directorio del resultado de la operación contendrá:

- El directorio *extracted_ data* en donde se encuentran los datos originales que fueron extraídos por la operación.
- Los archivos *inspected_ data.xml* y *source_ data.xml*.

4.3. Arquitectura

El sistema cuenta con un único punto de entrada y salida de datos a través de un componente al que denominamos *Coordinator*. El mismo ofrece todos los comandos con los que dispone el usuario. A su vez, el *Coordinator* se comunica con, o bien el *Operations Manager*, para ejecutar operaciones o consultar información sobre las mismas; o bien el *Extensions Manager*, para modificar las extensiones con las que cuenta el sistema.

Luego, por un lado podemos ver que el *Operations Manager* utiliza el *Definitions Database Manager* para obtener la información necesaria sobre las operaciones y con el *Repositories Manager* para obtener el extractor e inspector correspondientes a la operación a ejecutar.

Por otro lado, vemos que el *Extensions Manager* utiliza el *Definitions Database Manager* para agregar o eliminar definiciones de operaciones, tipos de datos y tipos de fuentes de datos. Luego, se comunica con el *Repositories Manager* para agregar o eliminar los archivos según dichas definiciones.

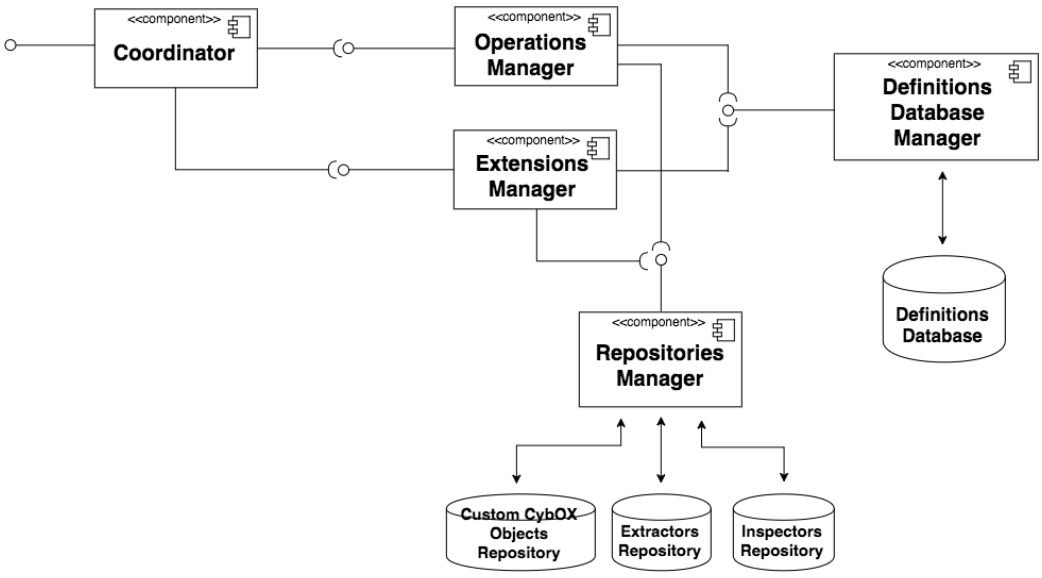

Figura 4.2: Diagrama de componentes del sistema

A continuación veremos en detalle cada uno de los componentes del sistema.

4.3.1. Coordinator

Se encarga de brindar en forma de fachada todos los comandos disponibles de la herramienta, los cuales permiten llevar a cabo los casos de uso vistos. Además, es el responsable de desplegar información al usuario en la ejecución de los mismos.

```
set_device_info(device_info: DeviceInfo)
list_operations(data_type: string, data_source: DataSource,
                device_info: DeviceInfo)
execute_operations(names: list(string), device_info: DeviceInfo)
add_ext(type: string, def_path: string)
rm_ext(type:string, name: string)
```

Interfaz Coordinator

Descripción de los métodos
Los métodos brindados corresponden a los comandos del sistema ya descritos en la sección de Casos de Uso.

4.3.2. OperationsManager

Se encarga de responder a consultas realizadas por el usuario sobre operaciones del sistema [Req. A2] y también de crear las instancias de las operaciones que son solicitadas por el Coordinator. Luego, el Coordinator se encarga de ejecutar dichas instancias de manera batch almacenando de forma organizada los resultados de cada operación [Req. A3].

```
get_operations_info(data_type: string, data_source: DataSource,
                    device_info: DeviceInfo): list(OperationInfo)
get_operation(name: string): Operation
```

Interfaz OperationsManager

Descripción de los métodos
El método `get_operations_info` obtiene la información de las operaciones que cumplen con los valores recibidos.

El método `get_operation` se encarga de crear una instancia de la operación correspondiente al nombre indicado y la devuelve.

4.3.3. DefinitionsDatabaseManager

Se encarga de facilitar la interacción con la base de datos de definiciones, de manera que ayuda a los componentes OperationsManager y ExtensionsManager a cumplir con los requerimientos [Req. A2], [Req. A3] y [Req. B].

```
query_operations_info(data_type: string, data_source: DataSource,
                      device_info: DeviceInfo): list(OperationInfo)
get_operation_info_by_id(id: integer): OperationInfo
get_data_type_custom_cybox_object_name(dt_name: string): string
get_data_source_type_extractor_name(dst_name: string): string
get_operation_inspector_name(op_name: string): string
get_operation_exec_info(name: string): string,string, dict(string)
exists_operation(name: string): bool
exists_data_type(data_type: string): bool
exists_data_source_type(data_source_type: string): bool
has_all_required_param_values(data_source: DataSource): bool
add_operation(name: string, data_type_id: integer,
              data_source_type_id: integer,
              inspector_id: string,
              param_values: dict(string),
              device_models: list(string),
              android_versions: list((string, string))
              ): bool
remove_operation(name: string): bool
add_data_type(name: string, cybox_object_name: string): bool
remove_data_type(name: string): bool
add_data_source_type(name: string,
                     extractor_name: string,
                     required_params: dict(string)
                     ): bool
remove_data_type(name: string): bool
```

Interfaz DefinitionsDatabaseManager

Descripción de los métodos
Los métodos `query_operations_info` y `get_operation_info_by_id` se encargan de realizar una consulta a la base de datos de definiciones para obtener la información de las operaciones que cumplen con los parámetros recibidos.

Los métodos `get_data_type_custom_cybox_object_name`, `get_data_source_type_extractor_name` y `get_operation_inspector_name` se encargan de obtener el nombre del módulo Python que implementa la extensión correspondiente.

El método `get_operation_exec_info` se encarga de obtener los datos necesarios para ejecutar la operación cuyo nombre coincida con el recibido por parámetro.

Los métodos `exists_operation`, `exists_data_type` y `exists_data_source_type` se encargan de consultar por la existencia de operaciones, tipos de datos y tipos de fuentes de datos respectivamente.

El método `has_all_required_param_values` se encarga de verificar que una fuente de datos cuenta con valores para todos los parámetros requeridos por el tipo de dicha fuente de datos.

Los métodos con prefijo `add` se encargan de agregar definiciones de operaciones, tipos de datos y tipos de fuentes de datos a la base de datos definiciones, mientras que los métodos con prefijo `remove` se encargan de eliminar las mismas de la base de datos.

4.3.4. ExtensionsManager

Se encarga de gestionar los puntos de extensibilidad del sistema de forma de satisfacer el requerimiento [Req. B].

```
add_ext(type: string, def_path: string)
rm_ext(type: string, name: string)
```

Interfaz ExtensionsManager

Descripción de los métodos
El método `add_ext` se encarga de validar la definición de la extensión recibida y añadirla al sistema, mientras que el método `rm_ext` es responsable de eliminar la extensión indicada.

4.3.5. RepositoriesManager

Se encarga de gestionar el almacenamiento de los archivos que contienen las clases utilizadas por las diversas extensiones del sistema. Para esto, maneja un repositorio para cada tipo de extensión. Además, tiene la responsabilidad de cargar en tiempo de ejecución clases a partir de sus archivos correspondientes almacenados en los repositorios y devolver instancias de las mismas.

```
add_file(repo_name: string, file_path: string)
remove_file(repo_name: string, file_name: string)
get_extractor_instance(name: string): Extractor
get_inspector_instance(name: string): Inspector
```

Interfaz RepositoriesManager

Descripción de los métodos
El método `add_file` recibe el nombre de un repositorio y la ruta al archivo a agregar, y se encarga de copiar dicho archivo al repositorio indicado.

El método `remove_file` recibe el nombre de un repositorio y el nombre del archivo a remover, y se encarga de buscar y eliminar del repositorio correspondiente al archivo indicado.

Los métodos `get_extractor_instance` y `get_inspector_instance` devuelven una instancia de la clase correspondiente.

4.4. Almacenamiento de datos

4.4.1. Repositorios

La herramienta cuenta con varios repositorios, o en otras palabras directorios, en donde se almacenan artefactos de las extensiones del sistema (aquellas definidas en Req. B). Concretamente se cuenta con tres repositorios:

1. **Custom CybOX objects**: Utilizado para almacenar las clases que implementan custom cybox objects para representar nuevos tipos de datos.
2. **Extractors**: Utilizado para almacenar las clases que implementan los extractors para cada tipo de fuente de dato.
3. **Inspectors**: Utilizado para almacenar las clases que implementan los inspectors para cada operación.

4.4.2. Base de datos

Utilizaremos una base de datos, a la cual llamaremos *DefinitionsDB*, para persistir las definiciones que describen a los diversos tipos de datos, tipos de fuentes de datos y operaciones disponibles en el sistema.

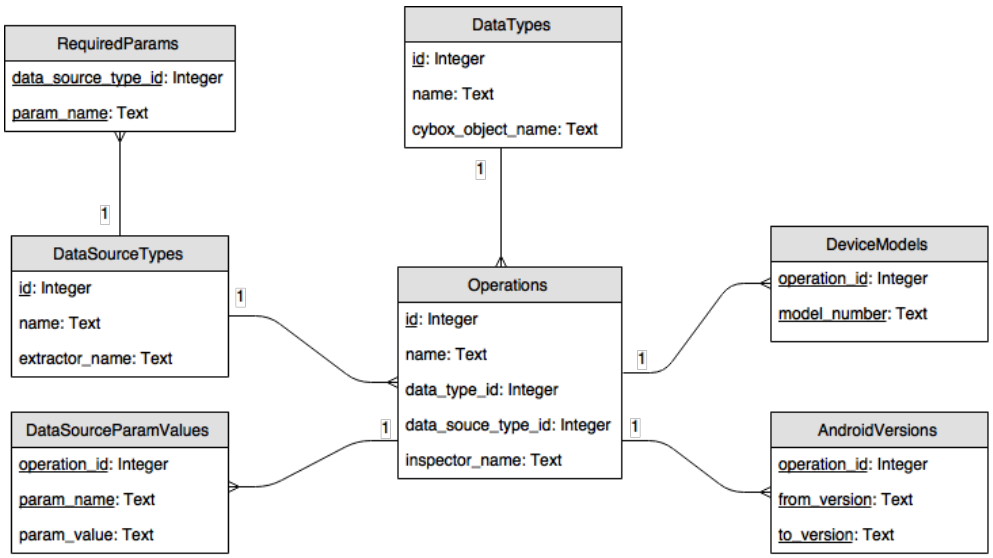

Figura 4.3: Esquema de la base de datos de definiciones

A continuación, veremos en detalle la información almacenada para cada una de las tablas de la base de datos de definiciones.

4.4.2.1. Tabla Operations

Columna	Tipo de dato	Descripción
id	*Integer*	*Identificador de la operación.*
name	*Text*	*Nombre de la operación.*
data_type_id	*Integer*	*Identificador del tipo de dato de la operación.*
data_source_type_id	*Integer*	*Identificador del tipo de la fuente de datos de la cual la operación extrae los datos.*
inspector_name	*Text*	*Nombre del inspector utilizado por la operación para examinar los datos extraídos.*

Cuadro 4.9: Tabla Operations

Esta tabla almacena el conjunto de operaciones del sistema. Es utilizada al momento de realizar la consulta que determina cuales son las operaciones que corresponden con el pedido del usuario. El identificador *id* de esta tabla es usado a nivel de la base de datos para identificar a la operación. Por otro lado, el campo *name* también identifica a la operación, pero es usado a nivel de usuario para consultar y ejecutar las mismas.

4.4.2.2. Tabla DataTypes

Columna	Tipo de dato	Descripción
id	*Integer*	*Identificador del tipo de datos.*
name	*Text*	*Nombre del tipo de datos.*
cybox_object_name	*Text*	*Nombre del objeto CybOX que representa el tipo de datos.*

Cuadro 4.10: Tabla DataTypes

Esta tabla almacena los tipos de datos soportados por el sistema.

4.4.2.3. Tabla DataSourceTypes

Columna	Tipo de dato	Descripción
id	*Integer*	*Identificador del tipo de fuente de datos.*
name	*Text*	*Nombre del tipo de fuente de datos.*
extractor_name	*Text*	*Nombre del extractor que implementa el mecanismo de extracción de datos para este tipo de fuente de datos.*

Cuadro 4.11: Tabla DataSourceTypes

Esta tabla almacena los tipos de fuentes soportados por el sistema. Junto al nombre de la fuente de datos, se guarda el nombre del script que realiza la extracción de dicha fuente. Este último nombre será utilizado para realizar la búsqueda del script en el momento de ejecutar una operación.

4.4.2.4. Tabla RequiredParams

Columna	Tipo de dato	Descripción
data_source_type_id	*Integer*	*Identificador del tipo de fuente de datos.*
param_name	*Text*	*Nombre del parámetro requerido por este tipo de fuente de datos.*

Cuadro 4.12: Tabla RequiredParams

Esta tabla almacena los parámetros requeridos de las fuentes de datos del sistema. Al agregar una nueva operación la cual utilice una fuente de datos existente, se valida que la nueva operación especifique valores para todos los parámetros de la fuente de datos.

4.4.2.5. Tabla DataSourceParamValues

Columna	Tipo de dato	Descripción
operation_id	*Integer*	*Identificador de la operación a la cual están asociados estos valores de parámetros.*
param_name	*Text*	*Nombre del parámetro.*
param_value	*Text*	*Valor del parámetro.*

Cuadro 4.13: Tabla DataSourceParamValues

Esta tabla almacena los valores de parámetros requeridos por el tipo de fuente de datos asociado a la operación indicada. Para ello, se indica para cada nombre de parámetro su correspondiente valor.

4.4.2.6. Tabla AndroidVersions

Columna	Tipo de dato	Descripción
operation_id	*Integer*	*Identificador de la operación a la cual está asociado este rango de versiones de Android soportadas.*
from_version	*Text*	*Versión inicial del rango.*
to_version	*Text*	*Versión final del rango.*

Cuadro 4.14: Tabla AndroidVersions

Esta tabla se utiliza para almacenar las versiones de Android soportadas por las operaciones. Cuando se agrega una nueva operación, se insertan nuevos registros en esta tabla con los rangos de las versiones de Android indicados.

4.4.2.7. Tabla DeviceModels

Columna	Tipo de dato	Descripción
operation_id	*Integer*	*Identificador de la operación a la cual está asociado este modelo de dispositivo soportado.*
model_number	*Text*	*Número del modelo del dispositivo soportado.*

Cuadro 4.15: Tabla DeviceModels

Esta tabla se utiliza para almacenar los distintos dispositivos soportados por las operaciones. Cuando se agrega una nueva operación, se insertan nuevos registros en esta tabla con los modelos de dispositivos indicados.

4.5. Descripción de los comandos

SetDeviceInfo

El comando permite indicarle al sistema la información del dispositivo que va a ser utilizado. Esto resulta muy útil al trabajar en el modo interactivo ya que evita que tengamos que indicar esta información como parámetro a los comandos `list` y `execute` cada vez que los deseamos utilizar. En caso que se le indique de todas formas como parámetro la información del dispositivo a estas operaciones y ya hayamos utilizado el comando `set_device_info`, la información indicada por parámetro será la considerada por el comando.

List

El comando es de utilidad al trabajar en modo interactivo para consultar las operaciones disponibles en el sistema y obtener información de las mismas antes de utilizarlas (Req. A2).

Los parámetros de este comando (*data_type*, *source_type*, *source_params*, *model* y *version*) actúan como filtros de búsqueda. Esto es, si a un parámetro se le indica un valor, se retornarán sólo aquellas operaciones que lo cumplen. En caso de dejar vacío un parámetro, no estamos imponiendo condición alguna sobre el mismo.

Execute

El comando es utilizado con el objetivo de ejecutar un conjunto de operaciones (Req. A3), tanto en modo interactivo como en modo batch. De esta forma, el comando recibe la lista de los nombres de las operaciones a ejecutar y la información del dispositivo. En caso de estar ejecutando en modo interactivo, la especificación de la información del dispositivo es opcional si el comando `set_device_info` fue ejecutado previamente.

Para cada operación se indica si la misma tuvo éxito o no. En caso de éxito, se indica el directorio en que fueron almacenados los datos resultantes de la operación, mientras que en caso de falla se indica el motivo de la misma.

AddExt & RmExt

El comando `add_ext` nos permite agregar extensiones al sistema, mientras que el comando `rm_ext` nos permite removerlas. Ambos reciben como parámetro el tipo de extensión (operación, tipo de dato o tipo de fuente de datos). En el caso de adición de una nueva extensión, el comando recibe una ruta al archivo de definición del mismo. En el caso de supresión de una extensión, el comando recibe el nombre de la extensión que deseamos remover.

Por más detalles sobre el funcionamiento de los comandos recién descritos, en el Apéndice B se encuentran los diagramas de flujo de cada comando.

4.6. Extensibilidad de la herramienta

Uno de los requerimientos fundamentales planteados [Req. B] es que la herramienta sea fácilmente extensible. Lo que buscamos con esto es permitir que la incorporación de nuevas operaciones se pueda realizar de forma habitual, de manera que sea posible contar con un conjunto de operaciones actualizado para enfrentar las necesidades del momento.

Para lograr esto, analizamos las etapas en que consiste una operación y buscamos modelar la misma de forma de facilitar la reutilización de partes que puedan ser compartidas. Esto permite reducir el esfuerzo requerido para desarrollar una nueva operación.

Dada la naturaleza del proceso que implica una operación (como vimos en la sección Conceptos Fundamentales), podemos distinguir las siguientes etapas:

1. Extracción de los datos.
2. Examinación de los datos extraídos.
3. Presentación de la información encontrada.

En la primera etapa, la operación extrae datos de una determinada fuente. Podemos notar que aquellas operaciones que obtienen datos de fuentes del mismo tipo (e.g. Aplicaciones), utilizan el mismo mecanismo (a menos de unos parámetros) para realizar la extracción. Este mecanismo es a lo que llamamos **extractor**.

En la segunda etapa, la operación examina los datos extraídos (en la etapa anterior) en busca de información de un cierto tipo de datos. Podemos notar que para este procedimiento es necesario que la operación conozca tanto la fuente de datos de donde provienen los datos extraídos como el tipo de dato que debe buscar. Esto lo hace una parte característica de la operación, y en consecuencia, no tiene sentido que sea reutilizado por otras operaciones. Al mismo le llamamos **inspector**.

En la tercera etapa, la operación organiza toda la información obtenida de forma de presentársela al usuario. Para esto, la información encontrada sobre los datos examinados es representada utilizando objetos CybOX y almacenada en archivos XML, mientras que los datos extraídos son almacenados en su forma original.

Ahora, vistas las distintas etapas que involucra una operación, vemos que

podemos modelar la misma de forma que conste de dos componentes fundamentales antes mencionados: un extractor y un inspector. Contar con esta separación nos brinda lo siguiente:

- En primer lugar, permite que operaciones que utilicen el mismo tipo de fuente de datos hagan uso de un mismo extractor. De esta forma, al desarrollar operaciones podemos reutilizar este componente.
- En segundo lugar, permite que el desarrollador de la operación no deba encargarse de organizar la información obtenida (tanto datos obtenidos por el extractor como datos examinados por el inspector), siendo la herramienta quien almacena de forma uniforme estos datos para todas las operaciones.

Con esto, acabamos de ver el punto de extensión más importante que brinda la herramienta, es decir, la posibilidad de extender el conjunto de operaciones con que cuenta la herramienta. Además, la herramienta también brinda otros dos puntos de extensión que permiten que podamos ampliar la diversidad de operaciones. Estos son:

- El conjunto de tipos de fuentes de datos. Este punto de extensibilidad permite que podamos definir operaciones que obtengan datos de nuevas fuentes de datos.
- El conjunto de tipo de datos. Este punto de extensibilidad permite que podamos definir operaciones que buscan, examinan y representan nuevos tipos de datos.

A continuación veremos en más detalle cada uno de estos tres puntos de extensión que brinda la herramienta.

4.6.1. Tipo de dato

La información obtenida por una operación sobre un tipo de dato es representada utilizando un objeto CybOX. Como hemos visto, el lenguaje CybOX cuenta con un conjunto de objetos predefinidos capaces de representar información sobre ciertos tipos de datos.

Si deseamos desarrollar una operación que examina un tipo de dato que CybOX es capaz de representar por defecto, la herramienta también lo soporta (al utilizar el módulo CybOX que nos provee esta funcionalidad). De esta forma, la base de datos es inicializada con las definiciones de los tipos de

datos correspondientes a los objetos predefinidos en CybOX.

Sin embargo, si deseamos desarrollar una operación que examina un tipo de dato que CybOX no soporta por defecto, será necesario crear un nuevo tipo de dato para representar esta información.

4.6.2. Tipo de fuente de datos

Como vimos, las fuentes de datos de un mismo tipo utilizan un mismo mecanismo de extracción de datos. El extractor es el responsable de implementar dicho mecanismo de extracción.

4.6.3. Operación

Una operación, además de extraer los datos, examina los mismos. Para eso, tiene asociado un inspector, que es quién implementa la examinación de los datos para el tipo de dato de la operación.

Capítulo 5

Implementación del prototipo

En este capítulo vamos a presentar varios de los aspectos más importantes que debimos considerar con respecto a las diversas tecnologías y herramientas que se han utilizado durante el proceso de desarrollo del prototipo de la herramienta *Android Inspector*.

5.1. Lenguaje de desarrollo

La elección de Python como lenguaje para el desarrollo del prototipo fue una de las primeras y más trascendentes decisiones que debimos tomar de cara a la implementación. Además, impactó en varios aspectos importantes de la herramienta. En particular, nos permitió brindar un lenguaje para desarrollar extensiones el cual fuera accesible.

Como veremos, si bien desde temprano el lenguaje se mostraba como candidato ideal para este desarrollo, la no experiencia de uso del mismo por parte de los autores era un factor a considerar dado el contexto del proyecto. Para mitigar el riesgo y asesorar los beneficios del lenguaje, nos familiarizamos en etapas previas con el mismo y realizamos pruebas de concepto en paralelo durante la etapa de diseño.

El amplio uso del cual goza Python por parte de la comunidad forense es un hecho conocido. Chet Hosmer, en su libro *Python Forensics* [13] dedica un excelente primer capítulo a describir cómo Python ayuda a cumplir con los desafíos que presenta el área forense y las características que presenta el lenguaje que lo hacen idóneo para este trabajo.

Como desafíos podemos encontrar:

- **La naturaleza cambiante de las investigaciones**: En la última década el foco del trabajo ha cambiado de simplemente extraer datos, recuperar archivos, etc a proveer análisis en tiempo real de redes, aplicaciones de cloud y dispositivos móviles así como la propia automatización de diversos análisis.
- **La creciente brecha entre desarrolladores e investigadores**: Los investigadores, examinadores, personal de respuesta de accidentes y auditores tienden a venir del campo de las ciencias sociales. En cambio, los desarrolladores de herramientas forenses suelen contar con una formación en el campo de la ciencia de la computación e ingeniería. La forma de proceder para la resolución de los problemas por parte cada uno de estos grupos suele ser bastante diferente.
- **El costo y disponibilidad de las herramientas**: El costo de las herramientas forenses puede llegar a ser abrumador, en especial cuando se tiene en cuenta el entrenamiento que las mismas requieren.

En cuanto a las características que nos ofrece Python, podemos destacar:

- **Soporte global**: Creado hace 35 años con la premisa fundamental de "programación para todos", Python es un lenguaje de propósito general en el cual es posible escribir código que sea entendible por no programadores.
- **Open source y plataforma independiente**: Dado que Python es open source, existen sólidas implementaciones de intérpretes Python para todas las plataformas incluidas Windows, Linux y Mac OS X.
- **Madurez del lenguaje**: Al día de hoy es uno de los lenguajes que goza de mayor adopción global. El lenguaje además cuenta con una amplia disponibilidad de bibliotecas, documentación y comunidad a la cual recurrir por más conocimiento.
- **Barrera de entrada**: Una de los factores de su éxito es la carencia de barreras de entrada que posee. Cuenta con entornos de desarrollo gratuitos, el lenguaje es independiente de la plataforma, el código es fácil de leer y escribir, y el soporte es vasto a nivel mundial.

Por otro lado, Brian Carrier, en su paper Open source Digital Forensics Tools [12], realiza un planteo interesante al examinar las reglas establecidas por los estándares de evidencia y comparar herramientas forenses de código abierto con aquellas de código cerrado. Una de sus conclusiones claves a las que llega

es que el software open source cumple de forma más clara y comprensiva los requerimientos establecidos por los lineamientos de pruebas de Daubert. Como referencia, el estándar de Daubert provee reglas de evidencias al nivel federal en los Estados Unidos.

Finalmente, si vamos al nivel más práctico también podemos observar la influencia en la comunidad forense que mencionamos. Las herramientas ofrecidas por MITRE en torno al proyecto CybOX se encuentran escritas en Python. En particular, la biblioteca que describiremos más adelante (python-cybox) únicamente está disponible en Python.

5.2. Entorno de desarrollo

A continuación se enumeran los detalles relevantes de los ambientes en los cuales se realizaron las pruebas del prototipo:

Sistemas operativos	Ubuntu 14.04 y Mac OS X 10.10
Intérprete Python	CPython 2.7.6
Dependencias Python adicionales	python-cybox (2.1.0.12) python-magic (0.4.6) python-tabulate (0.7.5) nose-parameterized (0.5.0) [Sólo si se desean correr las pruebas automáticas]
Dependencias de herramientas externas	adb (incluído en las platform-tools del Android SDK) aapt (incluído en las build-tools del Android SDK)
Variables de entorno	Se deben agregar a la variable PATH las rutas a los directorios platform-tools y build-tools correspondientes al Android SDK.

Cuadro 5.1: Entorno de desarrollo del prototipo

Esta información debe ser tomada en cuenta como requisitos de referencia para correr el prototipo.

Para instalar las bibliotecas Python requeridas de forma automática brindamos un archivo *requirements.txt* que declara todos los paquetes Python

requeridos por la aplicación. Utilizando el siguiente comando bajo el directorio de la aplicación podemos instalar de forma automática las mismas:

```
pip install -r requirements.txt
```

Cabe mencionar que si bien la implementación fue probada tanto en Linux como Mac OS X, la aplicación no presenta ningún impedimento para correr en Windows, dado que tanto Python como las herramientas del SDK de Android mencionadas se encuentran disponibles para los tres sistemas operativos.

Por otro lado, durante el desarrollo se hizo uso de una variedad de herramientas con el fin de asistir el desarrollo. Mencionamos las más significativas a continuación:

Herramienta	Nombre	Descripción
IDE	PyCharm Community Edition 4.5	Entorno que brinda muchas facilidades para el desarrollo de aplicaciones Python.
Emulador Android	Genymotion 2.3.1	Permite emular dispositivos x86 con diversas versiones de Android.
Manejador de paquetes de Python	pip 1.5.6	Permite obtener e instalar de forma sencilla módulos adicionales de Python.
Visualizador de observables CybOX	STIX-to-HTML 1.0.2	Herramienta que facilita la visualización de documentos CybOX y STIX.
Navegador de bases de datos SQLite	SQLiteBrowser 3.7.0	Interfaz gráfica que facilita examinar el esquema y el contenido de bases de datos SQLite.

Cuadro 5.2: Herramientas utilizadas en el desarrollo del prototipo

5.3. Biblioteca python-cybox

MITRE provee esta biblioteca [24] para Python con el fin de facilitar las tareas de parsing, manipulación y generación de contenido CybOX. La misma se plantea tanto permanecer fiel al estándar CybOX como al uso de buenas prácticas que son habituales en Python.

La biblioteca cuenta con activo desarrollo el cual sigue de cerca los últimos cambios del lenguaje. La forma en que está versionada la misma refleja la versión de CybOX que soporta. De esta forma, su esquema de versión es el siguiente: `major.minor.update.revision`, en dónde los primeros tres números corresponden a la versión de CybOX y el último es utilizado para indicar nuevas liberaciones de la propia biblioteca.

En cuanto a la arquitectura del código, podemos ver que cuenta con dos niveles de APIs:

- Por un lado, provee *bindings* a las construcciones CybOX definidas en el estándar. Las mismas son generadas en su mayoría de forma automática (utilizando la biblioteca de Python *generate_ds* [14]) a partir de los *XML schemas definitions* que definen formalmente al lenguaje CybOX. Esto representa un proceso equivalente al que podemos realizar utilizando JAXB [35] en el lenguaje Java.
- Por otro lado, también provee APIs de más alto nivel que permiten trabajar con las diversas construcciones disponibles en CybOX (como observables, objetos, etc) de forma natural en Python.

De esta forma, podemos trabajar con objetos de alto nivel como pueden ser *SMSMessageObject* ó *FileObject*, establecer relaciones entre ellos, introducirlos en un conjunto de observables y luego serializarlos a XML (gracias a las bindings) en tan sólo unas pocas líneas de código Python.

En cuanto a la posibilidad de extender los objetos que brinda el lenguaje, CybOX define un tipo de objeto al cual denomina *CustomObject* y es soportado por esta biblioteca. El mismo brinda la posibilidad de que podamos representar objetos que no se encuentran definidos en el estándar. De esta forma, este mecanismo nos permite extender la capacidad de representación de datos que brinda CybOX más allá de los objetos predefinidos con los que cuenta. Este aspecto resulta fundamental dado que permite a la herramienta soportar nuevos tipos de datos.

5.4. Interfaces de las extensiones

Veamos las interfaces que brindamos a los desarrolladores de extensiones:

```
class Extractor(object):
    __metaclass__ = ABCMeta
```

```
    @abstractmethod
    def execute(self, extracted_data_dir_path, param_values):
        """
        :type extracted_data_dir_path: string
        :type param_values: dict(string, string)
        :rtype : None
        """
        pass

class Inspector(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def execute(self, device_info, extracted_data_dir_path):
        """
        :type device_info: DeviceInfo
        :type extracted_data_dir_path: string
        :rtype : (list(Object), list(FileObject))
        """
        pass
```

Interfaces brindadas a los desarrolladores

Como Python no cuenta con un mecanismo para definir formalmente interfaces en el lenguaje (como lo brindan otros lenguajes como Java), decidimos implementar ciertos controles para tener certeza de que las implementaciones de extensiones respetan los contratos de interfaces establecidos.

Para esto, comenzamos por utilizar el módulo *ABC (Abstract Base Class)* que brinda la biblioteca estándar de Python. El mismo nos permite declarar clases abstractas, utilizando el atributo *__metaclass__* para indicar esto y declarando los métodos abstractos utilizando la anotación *abstractmethod*, como se puede observar en el cuadro anterior.

El mecanismo mencionado permite asegurarnos de dos aspectos sobre las clases que deseamos exponer como interfaces. En primer lugar, que las mismas no puedan ser instanciadas directamente, y en segundo lugar, que las clases que implementen la interfaz (esto es, hereden de esta clase) implementan efectivamente todos los métodos anotados. Estos controles son realizados por parte del módulo ABC en tiempo de ejecución. En caso de que una clase viole alguno de estos aspectos, se produce una excepción al intentar crear una instancia de la misma.

Además, dado que en Python un módulo (esto es, un archivo con extensión .py) puede contener múltiples clases de diversos nombres, y cada nueva extensión de la herramienta debe ser implementada en un nuevo módulo, deseamos realizar dos controles más:

- Por un lado, que el nombre de la clase que implementa la interfaz coincida con el nombre del módulo convertido a *Camel Case* (notar que los nombres de los módulos en Python deben estar en *Snake Case*). Esto se realiza simplemente como convención para facilitar la carga dinámica de las extensiones.
- Por otro lado, verificamos que la clase mencionada en el punto anterior implemente la interfaz correspondiente (esto es, herede ya sea de Extractor ó Inspector).

5.5. Carga dinámica de las extensiones

Antes de instanciar una operación, la herramienta primero obtiene las instancias del extractor e inspector correspondientes a la misma, utilizando el *RepositoriesManager*. Para esto, el *RepositoriesManager* cuenta con una función para uso interno denominada `get_class_from_file` que carga en tiempo de ejecución la clase a partir del archivo que le es indicado. Veamos los pasos que realiza esta función para cargar en memoria la clase:

- El primer paso es utilizar el método `import_module` del módulo `importlib` [38] para importar el módulo correspondiente al nombre del extractor o inspector. Dicho módulo es buscado en el repositorio correspondiente.
- El segundo paso es utilizar la función built-in de Python `getattr` que nos permite obtener la definición de un objeto dentro de un módulo. En este caso, lo utilizamos para obtener la clase del nombre que nos especificaron en el módulo que importamos en el paso anterior.
- Finalmente, el último paso es asegurarnos que dicha clase implementa la interfaz correspondiente (Extractor o Inspector) como vimos en la sección anterior y devolver la misma.

De esta forma, vemos como Python nos permite fácilmente realizar la carga a demanda de módulos. Esto permite que la herramienta sea capaz de manejar repositorios muy grandes a la vez que utiliza un consumo de memoria bajo ya que únicamente debe mantener en memoria el extractor e inspector de la operación que se encuentra ejecutando.

5.6. Manejo de errores

Para el manejo de errores decidimos hacer uso de excepciones, tanto para el código interno de la herramienta como para la forma que establecimos que las extensiones deben reportar sus errores de ejecución.

Para el primer caso, utilizamos excepciones estándar brindadas por el lenguaje, mientras que para el segundo caso definimos una excepción propia, la cual llamamos *OperationError*. Como se describe en el Apéndice A, esta es la forma que tanto extractors como inspectors deben utilizar para reportar sus errores.

5.7. Proceso de desarrollo y verificación

Con el fin de verificar el correcto funcionamiento del prototipo, a medida que implementamos los componentes fuimos desarrollando también test unitarios para cada uno.

Para el desarrollo del prototipo, seguimos un enfoque bottom up, esto es, comenzamos implementando los componentes *DefinitionsDatabaseManager* y *RepositoriesManager* que se observan en la figura 4.2 y verificando su funcionamiento mediante tests unitarios. Luego, implementamos *OperationsManager*, verificamos su funcionamiento utilizando tests unitarios que utilizaban mocks de sus dependencias (esto es, de *DefinitionsDatabaseManager* y *RepositoriesManager*) y luego además test de integración utilizando los componentes dependientes verdaderos. Para los restantes componentes procedimos de forma similar.

Finalmente, una vez que la capa de presentación, esto es el parser de comandos, se encontraba finalizada, realizamos test funcional exploratorio del prototipo.

5.8. Descripción de los paquetes

A continuación vemos una descripción de los distintos paquetes con los que cuenta la herramienta.

Nombre	Descripción
components	Contiene los módulos que conforman a los componentes principales de la herramienta descritos en la sección 4.3.
model	Contiene las clases referentes al modelo de datos descritas en la sección 4.2.
repositories	Contiene los tres repositorios de extensiones de la herramienta descritos en la sección 4.4.1.
test	Contiene las pruebas unitarias y de integración de los componentes descritas en la sección 5.7, así también como recursos utilizados para las mismas.
util	Contiene módulos utilitarios para facilitar el desarrollo de extractors e inspectors descritos en A.4.

Cuadro 5.3: Paquetes del prototipo

Capítulo 6

Caso de Estudio

En este capítulo vamos a analizar una serie de operaciones que desarrollamos con el fin de mostrar varios de los aspectos claves que brinda la herramienta. Para ello, planteamos el estudio de diversos tipos y fuentes de datos de aplicaciones de Mensajería y Redes Sociales dado que estas dos categorías representan hoy en día una proporción muy importante de la actividad promedio de los usuarios de dispositivos móviles.

Para la categoría de Mensajería, estudiaremos los tipos de datos Email y SMS utilizando como fuentes de datos las aplicaciones de Email y SMS que brinda Android por defecto.

Para la categoría de Redes Sociales, estudiaremos el tipo de dato *Contact* utilizando como fuentes de datos las aplicaciones de Facebook, WhatsApp y la agenda de contactos de Android.

De esta forma, en la primera mitad del capítulo veremos diversos aspectos importantes que se deben tener en cuenta al implementar operaciones y varias formas en que la herramienta facilita el desarrollo de las mismas. Luego, en la segunda mitad describiremos los conjuntos de datos utilizados para probar las operaciones y veremos un ejemplo de ejecución de las mismas.

6.1. Mensajería

Consideremos las operaciones antes descritas para datos de tipo SMS e Email. En el caso de ambas, hay un par de observaciones importantes a hacer:

- CybOX provee objetos predefinidos que permiten representar a sus tipos de datos (*SMSMessage* e *EmailMessage* respectivamente).
- Podemos utilizar el *ApplicationExtractor* que es provisto por la herramienta por defecto (como se menciona en el Apéndice A) dado que nuestro interés se encuentra en datos de aplicación.

De esta forma, si tomamos en cuenta los pasos que se enumeran en el Apéndice A, los únicos componentes que debemos implementar para dichas operaciones son sus respectivos inspectors.

El inspector de SMS representa un caso sencillo, por lo cual nos servirá para describir los aspectos básicos que debemos tener en cuenta al desarrollar un inspector. Mientras que en el caso del inspector de Email veremos algunos aspectos más avanzados que agrega.

6.1.1. Inspector de SMS

Este inspector extrae el tipo de dato *SMSMessage* definido en CybOX. La fuente de datos considerada es la aplicación por defecto de SMS de Android. A continuación veamos cuales son los puntos que tuvimos en cuenta para su implementación.

Para obtener la información sobre los SMS, debemos acceder a una base de datos SQLite (en este caso la misma se llama *mmssms.db*). Para esto, podemos hacer uso del módulo `sqlite3` disponible en las bibliotecas estándar de Python.

Los campos del objeto *SMSMessage* que utilizamos para representar los datos disponibles en la aplicación de SMS de Android fueron los siguientes:

Nombre	Descripción
Sender_Phone_Number	*Número del emisor.*
Recipient_Phone_Number	*Número del receptor.*
Sent_DateTime	*Número del destinatario.*
Body	*Fecha de envío.*
Length	*Cantidad de caracteres del mensaje.*

Cuadro 6.1: Campos utilizados del objeto SMSMessage

Con respecto a los números del emisor y receptor del SMS, estos no son almacenados explícitamente como tales por parte de la aplicación. La misma almacena el número del dispositivo con el que interactúa. Esto es, en caso de un SMS saliente se guarda el número de quién lo recibe y en caso de un SMS entrante se guarda el número del emisor. La aplicación no almacena el número del dispositivo en la cual opera.

Por otro lado, recordemos que todo inspector debe retornar un conjunto de objetos (que llamamos source objects) que representen los archivos de los cuales se obtuvo la información de los objetos examinados. Para simplificar esta tarea desarrollamos un método llamado `create_file_object` en una clase auxiliar. El mismo puede ser utilizado por cualquier inspector pasándole una ruta a un archivo y obteniendo como resultado un objeto CybOX File que contiene la información sobre el mismo.

En este caso, el inspector de SMS utilizará como archivo de fuente de datos únicamente al archivo de base de datos (*mmssms.db*). Veamos la representación XML del objeto File que obtenemos como resultado de utilizar el método auxiliar antes mencionado:

```
<cybox:Observable id="example:Observable-26">
    <cybox:Object id="example:File-3">
        <cybox:Properties xsi:type="FileObj:FileObjectType">
            <FileObj:File_Name>mmssms.db</FileObj:File_Name>
            <FileObj:File_Path>
/data/data/com.android.providers.telephony/databases/mmssms.db
            </FileObj:File_Path>
            <FileObj:File_Extension>.db</FileObj:File_Extension>
            <FileObj:Size_In_Bytes>102400</FileObj:Size_In_Bytes>
            <FileObj:File_Format>
                SQLite 3.x database, user version 57
            </FileObj:File_Format>
            <FileObj:Hashes>
                <cyboxCommon:Hash>
                    <cyboxCommon:Type
                        xsi:type="cyboxVocabs:HashNameVocab-1.0">
                        SHA256
                    </cyboxCommon:Type>
                    <cyboxCommon:Simple_Hash_Value>
a75af1ff214920c74aea4958e683c17558baa7464be316e5ecc82eb9296ae707
                    </cyboxCommon:Simple_Hash_Value>
                </cyboxCommon:Hash>
            </FileObj:Hashes>
        </cybox:Properties>
    </cybox:Object>
</cybox:Observable>
```

Finalmente, deseamos indicar los archivos de donde fue extraída la información presente en cada uno de los objetos examinados. De este modo, a cada

objeto *SMSMessage* creado se le agrega en sus RelatedObjects las referencias a los objetos de los cuales se obtuvo la información. Esto el inspector lo puede realizar utilizando el método `add_related` del objeto CybOX de la siguiente forma:

```
sms.add_related(source_objects[0],
                ObjectRelationship.TERM_EXTRACTED_FROM,
                inline=False)
```

La representación en XML de uno de los objetos *SMSMessage* producidos se ve de la siguiente forma:

```
<cybox:Observable id="example:Observable-18">
    <cybox:Object id="example:SMSMessage-1">
        <cybox:Properties xsi:type="SMSMessageObj:SMSMessageObjectType">
            <SMSMessageObj:Sender_Phone_Number>
                099555236
            </SMSMessageObj:Sender_Phone_Number>
            <SMSMessageObj:Sent_DateTime>
                2015-02-21T17:39:28
            </SMSMessageObj:Sent_DateTime>
            <SMSMessageObj:Body>Hola, todo bien?</SMSMessageObj:Body>
            <SMSMessageObj:Length>16</SMSMessageObj:Length>
        </cybox:Properties>
        <cybox:Related_Objects>
            <cybox:Related_Object idref="example:File-3">
                <cybox:Relationship
                    xsi:type="cyboxVocabs:ObjectRelationshipVocab-1.1">
                    Extracted_From
                </cybox:Relationship>
            </cybox:Related_Object>
        </cybox:Related_Objects>
    </cybox:Object>
</cybox:Observable>
```

6.1.2. Inspector de Email

Este inspector extrae el tipo de dato *EmailMessage* definido en CybOX. La fuente de datos considerada es la aplicación por defecto de Email de Android. Los campos del objeto *EmailMessage* que utiliza el inspector para representar los datos disponibles en la aplicación son los siguientes:

Nombre	Descripción
Header	*Cabezales del email.*
Raw Body	*Cuerpo completo en crudo.*

Attachments	*Referencias a los archivos adjuntos del email.*

Cuadro 6.2: Campos utilizados del objeto EmailMessage

A diferencia del caso anterior, ahora existen dos bases de datos, *EmailProvider.db* e *EmailProviderBody.db*, a las cuales debemos acceder para obtener información sobre los correos. En la primera se encuentran almacenados los cabezales de los mismos, mientras que en la segunda se encuentran almacenados sus cuerpos. A su vez, los correos también pueden tener archivos adjuntos asociados los cuales se encuentran almacenados en otros directorios (según se describe en [8]).

Para cada objeto *EmailMessage* que representa un correo, el inspector debe agregar los objetos que representan archivos adjuntos de dicho email al campo *attachements* (Ver cuadro 6.2). Estos objetos también deben ser agregados al conjunto de source objects que devuelve el inspector.

Al conjunto de source objects que devuelve el inspector debemos agregar tanto los objetos que representan las dos bases de datos utilizadas como todos los objetos que representan los archivos adjuntos de los emails representados.

De esta forma, un objeto *EmailMessage* podrá contar con más de un objeto relacionado. El primero será el objeto que referencia al archivo de la base de datos, *EmailProvider.db*, que contiene los cabezales de los correos. En caso que de que email contenga un cuerpo, el segundo objeto relacionado es el archivo que representa la base de datos *EmailProviderBody.db*, la cual contiene los cuerpos de los correos.

A continuación se muestra la representación en XML de uno de los objetos *EmailMessage*:

```
<cybox:Observable id="example:Observable-110">
  <cybox:Object id="example:EmailMessage-8">
    <cybox:Properties xsi:type="EmailMessageObj:EmailMessageObjectType">
      <EmailMessageObj:Header>
        <EmailMessageObj:To>
          <EmailMessageObj:Recipient
    xsi:type="AddressObj:AddressObjectType"category="e-mail">
            <AddressObj:Address_Value>
                to@gmail.com
            </AddressObj:Address_Value>
          </EmailMessageObj:Recipient>
        </EmailMessageObj:To>
```

```
        <EmailMessageObj:From xsi:type="AddressObj:AddressObjectType"
            category="e-mail">
          <AddressObj:Address_Value>
            Google &lt;no-reply@accounts.google.com&gt;
          </AddressObj:Address_Value>
        </EmailMessageObj:From>
        <EmailMessageObj:Subject>
            New sign-in from Firefox on Windows
        </EmailMessageObj:Subject>
        <EmailMessageObj:Date>
            2015-06-10T20:16:25
        </EmailMessageObj:Date>
        <EmailMessageObj:Message_ID>
    &lt;Y6KoAJHcll5r46_XiM1UcQ@notifications.google.com&gt;
        </EmailMessageObj:Message_ID>
        <EmailMessageObj:Content_Type>
          text/html
        </EmailMessageObj:Content_Type>
      </EmailMessageObj:Header>
      <EmailMessageObj:Raw_Body>
        -- El cuerpo del mensaje fue removido para
        -- mejorar la lectura de la estructura.
      </EmailMessageObj:Raw_Body>
      <EmailMessageObj:Attachments>
        <EmailMessageObj:File object_reference="example:File-85"/>
        <EmailMessageObj:File object_reference="example:File-87"/>
        <EmailMessageObj:File object_reference="example:File-89"/>
        <EmailMessageObj:File object_reference="example:File-91"/>
        <EmailMessageObj:File object_reference="example:File-93"/>
      </EmailMessageObj:Attachments>
    </cybox:Properties>
    <cybox:Related_Objects>
      <cybox:Related_Object idref="example:File-3">
        <cybox:Relationship
            xsi:type="cyboxVocabs:ObjectRelationshipVocab-1.1">
          Extracted_From
        </cybox:Relationship>
      </cybox:Related_Object>
      <cybox:Related_Object idref="example:File-73">
        <cybox:Relationship
            xsi:type="cyboxVocabs:ObjectRelationshipVocab-1.1">
          Extracted_From
        </cybox:Relationship>
      </cybox:Related_Object>
    </cybox:Related_Objects>
  </cybox:Object>
</cybox:Observable>
```

De forma recíproca, a los objetos que representan archivos adjuntos le agregamos como objeto relacionado al objeto *EmailMessage* dentro del cual venía el mismo. Esto el inspector lo realiza de la siguiente forma:

```
att.add_related(email,
                ObjectRelationship.TERM_CONTAINED_WITHIN,
                inline=False)
```

Veamos un ejemplo de un objeto CybOX que representa un archivo adjunto en su representación XML:

```
<cybox:Observable id="example:Observable-144">
  <cybox:Object id="example:File-85">
    <cybox:Properties xsi:type="FileObj:FileObjectType">
      <FileObj:File_Name>2</FileObj:File_Name>
      <FileObj:File_Path>
/data/data/com.android.email/databases/1.db_att/2
      </FileObj:File_Path>
      <FileObj:File_Extension/>
      <FileObj:Size_In_Bytes>
        1692
      </FileObj:Size_In_Bytes>
      <FileObj:File_Format>
        PNG image data, 144 x 144,
        8-bit/color RGBA, non-interlaced
      </FileObj:File_Format>
      <FileObj:Hashes>
        <cyboxCommon:Hash>
          <cyboxCommon:Type xsi:type="cyboxVocabs:HashNameVocab-1.0">
            SHA256
          </cyboxCommon:Type>
          <cyboxCommon:Simple_Hash_Value>
c6d6285d9fdc4074c366ef0544f1b6b23d7604183f74c8b9cc6d0cc56b1db4c0
          </cyboxCommon:Simple_Hash_Value>
        </cyboxCommon:Hash>
      </FileObj:Hashes>
    </cybox:Properties>
    <cybox:Related_Objects>
      <cybox:Related_Object idref="example:EmailMessage-8">
        <cybox:Relationship
            xsi:type="cyboxVocabs:ObjectRelationshipVocab-1.1">
          Contained_Within
        </cybox:Relationship>
      </cybox:Related_Object>
    </cybox:Related_Objects>
  </cybox:Object>
</cybox:Observable>
```

6.2. Redes sociales

Las aplicaciones de redes sociales suelen manejar un gran volumen de datos personales, lo cual las hace una fuente de datos muy interesante. Desafortunadamente, CybOX no cuenta con objetos predefinidos que puedan representar adecuadamente a muchos de los tipos de datos que solemos encontrar en las mismas. Frente a esta situación, recordemos que contamos con la alternativa de extender las capacidades de CybOX diseñando un nuevo objeto CybOX para representar un nuevo tipo de dato.

De esta forma, analizamos varias aplicaciones de redes sociales con el obje-

tivo de encontrar un tipo de datos lo suficientemente interesante que fuera utilizado por las mismas. Observamos que uno de los conceptos en común que manejan es el de contacto. Si bien el nombre utilizado para el mismo varía dependiendo del contexto de cada una (amigo, vínculo, seguidor, etc), éste forma parte imprescindible en todas ellas. Por lo tanto, decidimos diseñar un nuevo objeto CybOX llamado *Contact* para representarlo.

Los datos asociados a un contacto en cada red social particular varían pero un cierto conjunto de datos básicos es compartido a través de todas ellas (incluso en aplicaciones fuera de esta categoría como es la agenda del teléfono).

En la siguiente tabla podemos observar los campos del objeto *Contact*:

Nombre	Descripción
Display name	*Nombre completo del contacto.*
First name	*Nombre del contacto.*
Last name	*Apellido del contacto.*
Phone number	*Teléfono del contacto.*
Email	*Email del contacto.*
Profile picture	*URI a la imagen de perfil.*
Birthday	*Fecha del cumpleaños con formato yyyy-mm-dd.*

Cuadro 6.3: Campos del objeto Contact

La forma en que debemos implementar un nuevo objeto CybOX se puede ver en detalle en la sección A.2 del Apéndice A.

A continuación veremos cómo el tipo de dato *Contact* se ajusta a diferentes aplicaciones. También veremos qué otros detalles, además de contar con un nuevo objeto CybOX, debimos tener en cuenta al implementar cada uno de los inspectors.

6.2.1. Inspector de Facebook

Debido a que la aplicación de Facebook ha variado mucho en los últimos años, nos encontramos con la necesidad de desarrollar un mecanismo para obtener la versión de la aplicación a partir de los datos extraídos de la misma.

Al utilizar el *ApplicationExtractor*, además de obtener el directorio de datos privados de la aplicación, también obtenemos el APK de la aplicación indicada. De esta forma, si examinamos el *AndroidManifest* contenido en el APK, podremos obtener la versión de la aplicación dado que esta información se encuentra declarada en el mismo mediante la propiedad *android:versionName* [4].

Para realizar esto hay un detalle que debemos resolver primero, y es el hecho de que el archivo *AndroidManifest* contenido en el APK se encuentra en formato binario. Para obtener datos del mismo, podemos utilizar la misma herramienta que es utilizada por la toolchain de Android para convertirlo de su formato original en XML a binario. Esto se realiza de la siguiente forma:

```
aapt dump badging Facebook.apk
package: name='com.facebook.katana' versionCode='11209847'
    versionName='35.0.0.48.273' platformBuildVersionName='5.0-1521886'
```

Vemos que entre los datos devueltos por la herramienta, se encuentra el número de versión.

De esta forma, podemos invocar dicha herramienta desde el inspector para obtener la versión de la aplicación a partir del APK que se encuentra en los datos extraídos, y decidir qué acciones tomar en base a este dato. Con el fin de facilitar el desarrollo mediante la reutilización de código de uso frecuente, este mecanismo decidimos incluirlo en el módulo *inspectors_helper* mediante el método `get_app_version_name`.

Veamos ahora algunos detalles de la estructura que presenta la aplicación de Facebook.

Al igual que las aplicaciones de SMS e Email ya vistas, la aplicación Facebook cuenta con una base de datos (de nombre *fb.db*) en la cual se encuentra toda la información que precisamos relevante a contactos (o como Facebook los llama en este caso, amigos).

En el caso de Facebook, contamos con información de todos los campos representados en el objeto *Contact* y la misma se encuentra almacenada en una única tabla (friends). Por lo tanto, la obtención de los datos de contactos de Facebook resulta muy directa.

Veamos para la representación en XML para un objeto *Contact* de Facebook:

```
<cybox:Observable id="example:Observable-26">
  <cybox:Object id="example:Contact-1">
    <cybox:Properties xsi:type="CustomObj:CustomObjectType"
      custom_name="Contact">
      <cyboxCommon:Custom_Properties>
        <cyboxCommon:Property name="display_name">
          Joe Bautista
        </cyboxCommon:Property>
        <cyboxCommon:Property name="first_name">
          Joe
        </cyboxCommon:Property>
        <cyboxCommon:Property name="last_name">
          Bautista
        </cyboxCommon:Property>
        <cyboxCommon:Property name="phone_number">
          +639157175318
        </cyboxCommon:Property>
        <cyboxCommon:Property name="email">
          jobau518@yahoo.com
        </cyboxCommon:Property>
        <cyboxCommon:Property name="profile_picture">
          http://profile.ak.fbcdn.net/hprofile-ak-ash2/
            49302_1337617745_5617_q.jpg
        </cyboxCommon:Property>
        <cyboxCommon:Property name="birthday">
          1947-5-18
        </cyboxCommon:Property>
      </cyboxCommon:Custom_Properties>
    </cybox:Properties>
    <cybox:Related_Objects>
      <cybox:Related_Object idref="example:File-3">
        <cybox:Relationship
          xsi:type="cyboxVocabs:ObjectRelationshipVocab-1.1">
          Extracted_From
        </cybox:Relationship>
      </cybox:Related_Object>
    </cybox:Related_Objects>
  </cybox:Object>
</cybox:Observable>
```

6.2.2. Inspectors de WhatsApp

La primera diferencia que podemos percibir con Facebook, es que WhatsApp no almacena información acerca de los emails ni fecha de cumpleaños de los contactos.

En segunda instancia, hay una pequeña diferencia en cuanto a la imagen de perfil. A diferencia de Facebook en el cual la base de datos contiene una referencia a una URL, WhatsApp almacena un identificador que hace referencia a un archivo local. Para esto, en el campo *profile_picture* utilizamos la ruta al archivo en el dispositivo prefijando el scheme *file://*.

6.2.3. Inspector de la agenda telefónica de Android

En el caso de la agenda telefónica, el único dato que no almacena la aplicación es la fecha de cumpleaños del contacto.

La única dificultad que presenta la implementación del inspector de este caso es el hecho de que cuenta con un esquema de base de datos bastante más complejo que los anteriores. Para poder obtener la información de un contacto, es necesario acceder a múltiples tablas *contacts*, *people*, *contact_ methods* y *phones*.

6.3. Un extractor alternativo

6.3.1. Motivación

El hecho de que las operaciones cuentan por separado con los mecanismos de extracción e inspección, otorga a la herramienta la flexibilidad en cómo implementar las operaciones. De esta forma, en el caso de la extracción de datos de aplicaciones podemos llegar a implementar un extractor alternativo al extractor, mencionado en el Apéndice A, que la herramienta brinda por defecto. Esto permite que si lo deseamos podamos cambiar operaciones que utilizan el extractor de datos de aplicaciones por defecto que brindamos, por este nuevo extractor.

¿Para qué implementar un extractor de datos de aplicaciones alternativo si los datos que obtiene son los mismos? La diferencia está en que el método que utilicemos para realizar la extracción nos puede brindar otros beneficios. En este caso, el *ApplicationExtractor* que brinda la herramienta por defecto tiene la particularidad que precisa acceso root al dispositivo.

Como vimos en la subsección 2.5.5 del capítulo Estado del Arte, el mecanismo del comando `adb backup` proporciona una excelente alternativa para obtener los datos privados de las aplicaciones sin necesidad de contar con root en el dispositivo. El único inconveniente que presenta este mecanismo es que las aplicaciones pueden decidir si desean permitir o no el backup de sus datos y deshabilitar de esta forma este mecanismo. De todas formas, en nuestra experiencia la mayoría de las aplicaciones esta funcionalidad se encuentra habilitada dado que éste es su valor por defecto.

6.3.2. Detalles de implementación

Al ejecutar el comando `adb backup`, el usuario debe interactuar con el dispositivo y aceptar que se realice el backup de los datos. Este comando empaqueta y comprime los datos encontrados dentro del directorio de la aplicación (junto con su APK si así se lo indicamos). El nombre de los directorios de bases de datos, shared preferences y archivos es cambiado por el comando de backup (Por ejemplo, el directorio *shared_prefs* pasa a llamarse *sp*).

En [32] se describe en detalle el formato utilizado para crear los archivos de backup de Android.

La implementación de este nuevo extractor, al cual llamamos *AdbBackupExtractor*, consta de tres etapas:

1. Mediante el comando `adb backup` se obtiene el archivo *backup.ab*.
2. A este archivo, primero le quitamos el cabezal (que corresponde a los primeros 24 bytes). Luego, descomprimimos su contenido utilizando la biblioteca estándar *zlib* de Python, obteniendo un archivo tar. Finalmente, desempaquetamos el contenido del archivo tar utilizando la biblioteca estándar *tarfile* de Python.
3. De forma de preservar los nombres de directorios que son utilizados por las aplicaciones debemos renombrarlos a sus nombre originales, como ya vimos.

6.4. Datos de prueba

Con el fin de obtener resultados significativos en las pruebas de las operaciones implementadas, buscamos utilizar datos de entrada que fueran representativos de un uso real de los dispositivos móviles. Para esto, utilizamos dos *disk images* de muestra que brinda Oxygen Forensics (en su sitio web [36]) con el objetivo de simular casos reales para evaluar el funcionamiento de su suite de software forense.

Las especificaciones de los dispositivos correspondientes a las imágenes obtenidas son las siguientes:

	Número de modelo	Versión de Android
Samsung Galaxy Mini	GT-S5570	2.2

HTC EVO 3D	X515m	2.3

Cuadro 6.4: Dispositivos de prueba utilizados

Las particiones obtenidas de cada dispositivo fueron *userdata* y *sdcard*.

Como observamos, ambas imágenes datan de ya hace un par de años si juzgamos por la versión de Android que disponían los dispositivos. Lo mismo sucede con las versiones de las aplicaciones móviles utilizadas. Es importante además tener en cuenta que únicamente contamos con los datos de las aplicaciones y no con los del sistema.

Dado que los extractors utilizados en el caso de estudio realizar la extracción de los datos directamente del dispositivo, debemos tener una forma de implantar los datos obtenidos en un dispositivo para luego realizar las pruebas con la herramienta.

La opción ideal sería poder recrear los sistemas originales de los dispositivos de los cuales obtuvimos los datos en una máquina virtual de forma que las aplicaciones funcionen tal cual lo hacían en sus dispositivos original. Esta resulta ser una tarea muy difícil, en particular por los hechos antes mencionados de que no contamos con los datos del sistema y tampoco con los APKs de las aplicaciones.

Optamos por utilizar un emulador con una imagen estándar de Android en la cual implantamos los datos en sus correspondientes directorios del filesystem. En particular, el emulador utilizado fue Genymotion [18] con la imagen de la versión de Android 4.3 provista por ellos. Cabe destacar que la imagen provista ya se encontraba con acceso root, por lo cual no debimos realizar ningún otro paso antes de implantar los datos.

Con el fin de automatizar este proceso, creamos un script Python que se encarga de implantar un *dataset*. Específicamente, le llamamos *dataset* a un archivo *tar* que contiene tanto APKs de aplicaciones como directorios correspondientes a los que se encuentran en la ruta */data/data* de un dispositivo. De esta forma, al especificar el nombre de uno de estos archivos el script toma los directorios contenidos en el archivo tar y los copia a la ruta */data/-data* del dispositivo e instala los apks de aplicaciones. Para ello simplemente utiliza los comandos `adb install` y `adb push`.

Los *datasets* con los que contamos corresponden a los antes descritos sólo que únicamente contienen los datos de las aplicaciones utilizadas en el caso de estudio, estas son Facebook, WhatsApp y las aplicaciones por defecto de email, SMS y contactos de Android. Además en el caso de las aplicaciones de terceros (Facebook y WhatsApp) contienen los APKs de sus últimas versiones.

6.5. Ejecución de la herramieta

Para terminar, veremos un ejemplo de ejecución interactiva de la herramienta en la cual primero utilizaremos el script visto en la sección anterior para cargar los datos en un emulador y luego realizaremos varias extracciones de datos utilizando varias de las operaciones que se desarrollaron a lo largo de este capítulo.

Comenzamos ejecutando el script `load_data_sets` utilizando el *dataset* correspondiente al dispositivo HTC Evo 3D adquirido.

```
$ ./load_data_sets.py --ds HTC_Evo_3D
Installing 'com.facebook.katana-1.apk' into the device...
Installing 'com.whatsapp-1.apk' into the device...
Pushing 'com.whatsapp' data into the device...
Pushing 'com.android.providers.telephony' data into the device...
Pushing 'com.facebook.katana' data into the device...
Pushing 'com.android.providers.contacts' data into the device...
The data set was pushed successfully.
```

Una vez que contamos con los datos cargados en el emulador, iniciamos la herramienta en modo interactivo de la siguiente forma:

```
$ ./andi.py
Android Inspector v1.0
```

El primer comando que vamos a utilizar es `set_device_info` para indicarle a *Android Inspector* el modelo del dispositivo emulado que se encuentra conectado y la versión de Android que está corriendo.

```
(Andi) set_device_info -m GT-I9300 -v 4.3
Device model 'GT-I9300' running Android version '4.3' was set as the
    current device information.
```

De esta forma, cuando utilicemos los comandos `list` y `execute`, la información del dispositivo recién establecida será tomada en cuenta de forma implícita tanto para filtrar por las operaciones soportadas en el caso del primero como para pasarle esta información a cada operación que ejecuta en el caso del segundo (recordemos que la misma es utilizada por los inspectors al ejecutar una operación).

Veamos la información de las operaciones que se encuentran disponibles para el dispositivo que establecimos utilizando el comando `list`.

```
(Andi) list
Name                          Data type
----------------------------  ------------
EmailMessageAOSPEmail         EmailMessage
SmsMessageAOSPSms             SMSMessage
ContactFacebook               Contact
ContactWhatsApp               Contact
ContactAOSPAgenda             Contact

Data Source
-------------------------------------------------------------
Application{package_name:com.android.email}
Application{package_name:com.android.providers.telephony}
Application{package_name:com.facebook.katana}
Application{package_name:com.whatsapp}
Application{package_name:com.android.providers.contacts}

Supported devices                Supported Ver.
-------------------------------------------------
[GT-I9300, XT1053]               [2.3.7-5.1.1]
[GT-I9300, XT1053]               [2.2.0-4.4.4]
[GT-I9300, Nexus5, XT1053]       [4.1-4.4.4]
[GT-I9300, XT1053]               [4.1-4.4.4]
[GT-I9300, XT1053]               [2.3-4.4.4]
```

Ahora que conocemos las operaciones que podemos ejecutar para este dispositivo podemos empezar a extraer los datos que deseamos. Por ejemplo, para extraer los mensajes de texto de la aplicación por defecto de mensajería de Android observamos que tenemos la operación *EmailMessageAOSPEmail* que desarrollamos en la sección 6.1.1. De esa forma, podemos ejecutarla pasándole el nombre de la misma al comando `execute`.

```
(Andi) execute -op SmsMessageAOSPSms

[1/1] Executing 'SmsMessageAOSPSms':
Fetching '/system/app/TelephonyProvider.apk' file...
Fetching '/data/data/com.android.providers.telephony' directory...
COMPLETED. Data stored to 'results/SmsMessageAOSPSms_20150826_081415'.

1 operation(s) completed successfully.
```

La salida del comando en este caso nos muestra que la operación que ejecutamos completó exitosamente. Además, nos indica el directorio en dónde quedaron almacenados tanto los datos extraídos como los archivos XML de CybOX conteniendo los datos examinados.

El comando `execute` permite ejecutar múltiples operaciones, una detrás de la otra, si especificamos varias. Por ejemplo, podemos ejecutar las operaciones *ContactFacebook* y *ContactWhatsApp* que desarrollamos en la sección 6.2 para extraer los datos de contactos de estas dos aplicaciones.

```
(Andi) execute -op ContactFacebook ContactWhatsApp

[1/2] Executing 'ContactFacebook':
Fetching '/data/app/com.facebook.katana-1.apk' file...
Fetching '/data/data/com.facebook.katana' directory...
COMPLETED. Data stored to 'results/ContactFacebook_20150826_084055'.

[2/2] Executing 'ContactWhatsApp':
Fetching '/data/app/com.whatsapp-1.apk' file...
Fetching '/data/data/com.whatsapp' directory...
COMPLETED. Data stored to 'results/ContactWhatsApp_20150826_084107'.

2 operation(s) completed successfully.
```

Utilizando el comando `tree` de Unix podemos observar la estructura de los directorios con los datos resultantes de las tres operaciones que ejecutamos.

```
$ tree -L 3
.
|__ ContactFacebook_20150826_084055
|   |__ extracted_data
|   |   |__ com.facebook.katana-1.apk
|   |   |__ databases
|   |__ inspected_data.xml
|   |__ source_data.xml
|__ ContactWhatsApp_20150826_084107
|   |__ extracted_data
|   |   |__ com.whatsapp-1.apk
|   |   |__ databases
|   |   |__ files
|   |   |__ shared_prefs
|   |__ inspected_data.xml
|   |__ source_data.xml
|__ SmsMessageAOSPSms_20150826_081415
    |__ extracted_data
    |   |__ app_parts
    |   |__ databases
    |   |__ shared_prefs
    |   |__ TelephonyProvider.apk
    |__ inspected_data.xml
    |__ source_data.xml
```

Sería interesante observar una porción de los archivos XML que contienen los objetos CybOX producidos por las operaciones. Para esto, podemos utilizar la herramienta `cybox-to-html` [29] que produce una representación de los XML más clara en HTML. A continuación veremos un par de extractos de los objetos CybOX que contienen los archivos *source_ data.xml* y *inspected_ data.xml* correspondientes a la ejecución de la operación *ContactFacebook*.

En el archivo *source_ data.xml* podemos encontrar el siguiente objeto File:

Figura 6.1: Objeto *File* que representa al archivo de base de datos *fb.db* del cual se extrajeron los datos sobre los contactos de Facebook.

En tanto, en el archivo *inspected_ data.xml* podemos encontrar una serie de objetos *Contact* representando a cada uno de los contactos de Facebook obtenidos. Veamos un par de ellos:

Figura 6.2: Información de los dos primeros objetos *Contact* del archivo *inspected_ data.xml*.

De esta forma, vimos una sesión completa de ejecución de la herramienta en la cual realizamos diversas extracciones de datos y examinamos su representación. Queda en manos del investigador forense cómo utilizar esta información expresada mediante el lenguaje CybOX para realizar diversos análisis a partir de la misma.

Capítulo 7

Trabajo a futuro

En este capítulo plantearemos algunas ideas que fueron surgiendo en diversas etapas del desarrollo del trabajo. Creemos que las mismas presentan buenas oportunidades para mejorar el diseño de la herramienta y la implementación del prototipo desarrollado.

7.1. Generalización de las condiciones impuestas por las operaciones

Como vimos, una operación cuenta con información que condiciona los dispositivos que soportada. Esta información en el prototipo desarrollado corresponde únicamente a modelos de dispositivos y versiones de Android. Sería interesante desarrollar un mecanismo extensible que permita generalizar este concepto para permitir que las operaciones puedan imponer *condiciones* sobre este tipo de información del dispositivo.

De esta forma, los modelos y versiones de Android podrían pasar a ser condiciones particulares que impone una operación, en lugar de atributos fijos de las mismas. En caso que deseemos expresar que una operación soporta únicamente un cierto modelo de dispositivo, la operación simplemente tendría que imponer una condición de este tipo con el valor que representa al modelo de dispositivo correspondiente. Si en cambio la operación no desea restringir su uso según el modelo del dispositivo, la misma puede optar por no imponer condición alguna de este tipo permitiendo así su uso en cualquier modelo de dispositivo.

Veamos dos tipos de condiciones nuevas que consideramos que sería verdaderamente útil contar con ellas. Se tratan de si el dispositivo cuenta con *acceso root* y de si el dispositivo tiene el *modo debug* activado. En el prototipo desarrollado el usuario es el responsable de verificar si la operación que va a utilizar requiere alguno de estos dos y si es así asegurarse que el dispositivo cumple con los mismos. En caso de que una operación pueda imponer este tipo de condiciones, la herramienta sería capaz de verificar que las mismas se cumplen al compararlas con la información del dispositivo ingresada por el usuario sobre el dispositivo.

Para establecer la información del dispositivo contamos con el comando `set_device_info`. Actualmente el mismo solicita modelo del dispositivo y versión de Android que corre. Al agregar condiciones, como las mencionadas anteriormente, este comando debe pasar a solicitar esta información para que luego las operaciones puedan utilizar condiciones en función de la misma.

El comando `list`, que actualmente permite filtrar operaciones según sus campos (incluyendo modelo de dispositivo y versión de Android) pasaría a permitir filtrar operaciones también según los diversos tipos de condiciones. No casualmente la forma en que implementamos en el prototipo dicho comando hace que agregar esta funcionalidad sea muy sencillo. Los cambios que se requieren son agregar una nueva clase que herede de la clase abstract *Filter* para cada tipo de condición (de forma similar a como hoy tenemos la clase *AndroidVersionFilter* para filtrar por versión de Android) y luego ajustar el módulo de parsing de la entrada de línea de comando.

Por último, debemos notar que el comando `execute` continuará recibiendo como parámetro un objeto del tipo *DeviceInfo* el cual contendrá la nueva información del dispositivo. Pensamos que sería razonable que esta información además de ser pasada al inspector como sucede actualmente, también aprovechemos a pasarla al extractor ya que podemos tener datos que le sean relevantes como si el dispositivo cuenta con el modo debug activado mencionado antes.

7.2. Automatización de la obtención de la información del dispositivo

El prototipo desarrollado cuenta con el comando `set_device_info` el cual es utilizado para ingresar manualmente la información del dispositivo, esto es, el modelo del mismo y la versión de Android que corre.

Dado que esta información resulta importante para la ejecución de las operaciones, la herramienta requiere que la misma sea especificada antes de poder ejecutar cualquier operación. Por lo tanto, la obtención de esta información es una tarea indispensable que siempre debemos realizar. De esta forma, consideramos que automatizar esta tarea facilitaría el uso de la herramienta.

Para implementar esto, podría añadirse un nuevo comando para que desempeñe la tarea de obtención de la información de forma automática y luego establezca la información del dispositivo llamando internamente al comando `set_device_info`.

Cabe destacar que este comando no tiene intención alguna de sustituir al comando `set_device_info`, sino que su finalidad es la obtención y establecimiento de dicha información de forma automática. Además, consideramos que es esencial que siempre podamos contar con la posibilidad de establecer esta información de forma manual dado que el mecanismo que implemente la obtención de los datos de forma automática con seguridad no sea infalible.

7.3. Caching de los datos extraídos de una fuente de datos

Como vimos en la sección 3.4, una operación extrae información de un tipo de dato de una determinada fuente de datos. Si en una sesión utilizamos varias operaciones que toman sus datos de la misma fuente de datos, la herramienta deberá realizar la misma extracción para cada una de ellas, debido a la forma en que se implementó el prototipo.

Este aspecto se puede solucionar si implementamos un mecanismo de caché para los datos de una misma fuente de datos que son obtenidos durante una sesión de ejecución de la herramienta.

Recordemos que las operaciones constan de dos componentes: extractor e inspector. Esto hace que el proceso de extracción de los datos sea totalmente independiente del de examinación de los mismos. Esta separación nos permite implementar el caché mencionado de forma transparente al usuario de la herramienta. La única diferencia que notará es que tras ejecutar una operación, las siguientes operaciones que utilicen la misma fuente de datos que esta última se realizarán más rápidamente.

De esta forma, si al ejecutar una operación ya se encuentra en caché los datos

extraídos de la fuente de datos de donde pretende obtener datos la operación, la herramienta en vez de ejecutar el extractor correspondiente simplemente realiza una copia de los datos del caché.

Por otro lado, este mecanismo no sólo brinda una mejora de rendimiento, sino que también hace posible disminuir la cantidad de accesos que debemos realizar al dispositivo ayudando a prevenir modificaciones innecesarias que se puedan hacer al mismo de forma inadvertida.

Por último, el hecho de realizar una copia de los datos extraídos a partir del caché en vez de utilizar esos directamente asegura que los datos que examina el inspector de una operación no fueron modificados previamente por otro inspector.

7.4. Mayores garantías en la preservación de los datos extraídos

Hay algunos procedimientos que se podrían incorporar a la herramienta para que la misma brinde mayores garantías desde el punto de vista forense para la conservación de la integridad de los datos extraídos.

En primer lugar, podríamos llevar a cabo dos etapas de verificación:

- Verificar que la integridad de los datos extraídos fue conservada al realizar la extracción de los mismos del dispositivo. Para esto, debemos verificar que el hash de los archivos obtenidos del dispositivo coincide con los datos extraídos.
- Verificar que los datos extraídos no son modificados al ser examinados por un inspector. Para esto, podríamos utilizar los hashes previamente obtenidos al realizar la extracción de los datos, computar el hash de los mismos tras finalizada la examinación y compararlos.

En segundo lugar, una posibilidad que tiene su trade-off interesante es la de limitar la libertad de los inspectors y obligarlos a utilizar un conjunto de primitivas (a ser desarrolladas). Entre las ventajas que brindaría esto se encuentra que podemos asegurar que el acceso a los archivos que realizan los inspectors sea de *sólo lectura* únicamente y también poder llevar un log de todas las acciones realizadas por un inspector.

7.5. Desarrollo de otros tipos de extractors

Los extractors desarrollados en este trabajo todos utilizan el protocolo *adb* para obtener los datos. El mismo nos resultó conveniente dado que permite interactuar de forma indiferente tanto con dispositivos virtuales (esto es, emuladores) como con dispositivos físicos (a través de USB).

Ahora bien, la única restricción con la cual fue diseñado el extractor fue que el mismo debe realizar una extracción lógica de la fuente de datos indicada y almacenar los archivos extraídos en el directorio que se le especifica. Por lo tanto, la elección de cómo extraer los datos queda en manos del desarrollador del extractor.

Dada esta libertad, una primera idea interesante a investigar sería la realización de extractors que obtengan datos del sistema operativo, en contraposición a los extractors que desarrollamos que obtienen datos de aplicaciones.

Otra idea que creemos que puede ser de sumo interés es el desarrollo de extractors que en vez de tomar datos a partir de un dispositivo conectado lo hacen a partir de una *disk image*. Esto resultaría importante dado que permite seguir de más cerca los lineamientos que suelen establecer los diversos procesos forense, esto es, el contar con un respaldo de los datos extraídos y luego trabajar sobre una copia de los mismos en vez de trabajar sobre el dispositivo original. Lo más importante aún es que esto perfectamente puede ser implementado en un nuevo extractor con el prototipo actual, sin necesidad de modificar la herramienta.

7.6. Añadir más datos del proceso al contenido CybOX generado

Actualmente la herramienta añade algunos datos como son el nombre y versión de la misma (esto es, Android Inspector v1.0), un listado de archivos extraídos (con su respectiva información y hashes) y en cada objeto CybOX de los datos examinados referenciamos a los archivos de los cuales se obtuvo su información.

Dado que CybOX permite representar otros datos de interés como pueden ser información sobre los colaboradores que llevaron a cabo el proceso e información sobre el dispositivo del cual se realizó la extracción, puede ser interesante contemplar la posibilidad de agregar este tipo de información.

7.7. Soporte para otras plataformas

Un aspecto que consideramos que podría resultar muy importante para la herramienta es el de agregar soporte para la extracción y examinación de datos de otros sistemas operativos móviles como iOS y Windows Phone.

Desde el punto de vista del diseño de la herramienta, en principio no serían necesarios cambios importantes. El principal cambio que deberíamos realizar es el de agregar un campo que indique el sistema operativo que soporta la operación y cambiar el nombre del campo *versiones de Android soportadas* por *versiones soportadas* para indicar las versiones del sistema operativo correspondiente que la misma soporta. De esta forma, el comando `set_device_info` además de requerir especificar el modelo del dispositivo y versiones del sistema operativo que soporta también deberá solicitar al usuario el sistema operativo del dispositivo.

El desafío para soportar nuevas plataformas viene por el lado del desarrollo de extractors que realicen el trabajo de obtener los datos. Para ello, se deberán investigar los mecanismos disponibles con los que se cuenta en cada plataforma y las condiciones necesarias en las que debe estar el dispositivo para poder obtener los datos.

Capítulo 8

Conclusiones

El área de *mobile forensics*, en la cual se enmarca este trabajo, actualmente se encuentra en un momento interesante. Si bien la misma aún no cuenta con el nivel de estandarización de procedimientos forenses con el que se dispone en *computer forensics*, en los últimos años han surgido una serie de iniciativas importantes que apuntan a la estandarización de varios de estos procedimientos. La tarea de extracción de datos de dispositivos móviles forma parte de dicho proceso y es en la que este trabajo estuvo enfocado con el fin de brindar solución a algunos de los desafíos que la misma presenta.

Una de las iniciativas interesantes que vimos es la llevada a cabo por MITRE, en colaboración con el *Department of Homeland Security* (DHS) de los EEUU. Esta tiene como objetivo el desarrollo de varios lenguajes que buscan estandarizar y automatizar la forma en que se realiza el intercambio de la información de seguridad en diversas áreas. Uno de dichos lenguajes es CybOX, del cual hace uso la herramienta que desarrollamos para representar información sobre los datos extraídos.

El lenguaje CybOX fue la pieza clave que nos permitió cumplir con el objetivo de desarrollar una herramienta que produjera información en un formato fácilmente consumible por otras herramientas. Esto se debe a su adopción como estándar abierto y la madurez de las herramientas que brinda. A su vez, un aspecto particularmente destacable fue la gran disposición que mostraron de parte de MITRE para aclarar aspectos que no se encontraban documentados sobre las capacidades de extensibilidad del lenguaje y de la biblioteca *python-cybox*. Esto fue importante ya que nos permitió validar que CybOX contaba con el poder de expresión y extensibilidad que buscábamos en el

lenguaje a ser utilizado en la herramienta.

Recordemos que el objetivo principal que nos planteamos para la herramienta fue el de facilitar la extracción de datos de dispositivos móviles. Para esto, diseñamos la herramienta con un nivel de abstracción que permitiera a un investigador forense prescindir del conocimiento de cómo es implementada cada operación de extracción de datos que desea realizar. Para esto, concebimos cada operación como un procedimiento que extrae cierto tipo de datos, de una determinada fuente de datos y soportando un determinado conjunto de dispositivos móviles. Eso permitió que la herramienta pudiera contar con una interfaz de entrada la cual permite especificar los datos que el investigador desea obtener en términos que éste conoce.

El tercer y último objetivo que nos planteamos para la herramienta fue el de facilitar la extensión de la misma. Para ello, hicimos un gran énfasis en el diseño de la arquitectura. Las consideraciones realizadas para contar con una herramienta altamente extensible y con un mecanismo que permitiera importar fácilmente las extensiones desarrolladas fueron aspectos que resultaron esenciales. Esto se debe a que para poder adaptarnos a los rápidos cambios que presenta el ecosistema de dispositivos móviles y escalar para soportar una porción significativa de los mismos, es necesario contar con una forma de colaborar con otros. De esta manera, si podemos encapsular y compartir el conocimiento técnico que desarrollamos para que pueda ser reutilizado por otros, estaremos ampliando la capacidad con la que contamos para obtener datos.

El caso de estudio realizado permitió ilustrar las principales funcionalidades de la herramienta y la forma de desarrollar cada uno de los diversos tipos de extensión de la herramienta. De forma de realizar un caso representativo del uso real de la herramienta, tomamos las áreas de mensajería y redes sociales para desarrollar las extensiones. Además, obtuvimos un par de conjuntos de datos de prueba elaborados para este fin. Cabe mencionar que la obtención de dichos conjuntos de datos de prueba no fue una tarea sencilla dada la escasa disponibilidad de los mismos que hay públicamente en comparación a lo que podemos estar acostumbrados para otras áreas de seguridad informática. Creemos que esto se puede deber tanto al trabajo que implica sanitizar los datos como el hecho que el área es relevantemente nueva.

En cuanto al prototipo, logramos desarrollar una primera versión de la herramienta que permite mostrar las funcionalidades principales que nos planteamos obtener y desarrollamos un conjunto de extensiones para la misma en el

área particular de datos de aplicaciones. El foco de su desarrollo estuvo puesto en contar con una arquitectura que luego permitiera continuar ampliando las funcionalidades de la herramienta en base a la visión que propusimos para la misma. En particular, vimos que hay varias ideas, vistas en el Capítulo 7, que presentan oportunidades interesantes para continuar con el desarrollo de la herramienta, como son el desarrollo de extractors para otros dominios de datos y el soporte para otras plataformas móviles.

A su vez, es importante mencionar el proceso de desarrollo que tuvimos al implementar el prototipo. Una de las decisiones más importantes vista en retrospectiva fue la elección del lenguaje Python. Desde el comienzo fue un claro candidato dada la preferencia del mismo por parte de la comunidad forense open source y por el hecho de que la única biblioteca de alto nivel que provee CybOX es en el lenguaje Python. Para el desarrollo exitoso del prototipo además nos resultó muy importante la práctica de acompañar el desarrollo de cada módulo con el desarrollo de tests para el mismo. Esto nos permitió iterar rápidamente al detectar regresiones con facilidad. La versión final del prototipo cuenta con 55 tests unitarios cubriendo 88 % del total de líneas de código escrito.

A modo de cierre, entendemos que este trabajo realiza un aporte en dos dominios complementarios. Por un lado, se realizó un estudio sobre el estado del arte de la extracción de datos de dispositivos móviles y las problemáticas que presenta dicha tarea. Por otro lado, en base a dicho estudio se desarrolló una herramienta para atacar varios de los problemas presentados. Como vimos, la herramienta fue desarrollada con un énfasis en el diseño de su arquitectura. De esta forma, otros interesados podrán extenderla a dominios de datos en el área de interés que deseen trabajar.

Bibliografía

[1] *Android Architecture*, http://www.tutorialspoint.com/android/android_architecture.htm, Última visita: 5 de setiembre de 2015.

[2] Anders O. Flaglien, et al., *Storage and exchange formats for digital evidence*, (2011).

[3] Android Developers, *Android Storage*, http://developer.android.com/guide/topics/data/data-storage.html, Última visita: 5 de setiembre de 2015.

[4] ———, *AndroidManifest, the versionName attribute*, http://developer.android.com/guide/topics/manifest/manifest-element.html#vname, Última visita: 5 de setiembre de 2015.

[5] ———, *Saving data safely*, http://android-developers.blogspot.com/2010/12/saving-data-safely.html, Última visita: 5 de setiembre de 2015.

[6] Android Open Source Project, *Android Debug Bridge Repository*, https://android.googlesource.com/platform/system/core.git/+/master/adb/, Última visita: 5 de setiembre de 2015.

[7] ———, *Android Security Background*, https://source.android.com/devices/tech/security/images/image00.png, Última visita: 5 de setiembre de 2015.

[8] ———, *AttachmentProvider implementation*, https://android.googlesource.com/platform/packages/apps/Email.git/+/master/provider_src/com/android/email/provider/AttachmentProvider.java, Última visita: 5 de setiembre de 2015.

[9] ———, *Codenames, Tags, and Build Numbers*, https://source.android.com/source/build-numbers.html#platform-code-names-

versions-api-levels-and-ndk-releases, Última visita: 5 de setiembre de 2015.

[10] Brian Neil Levine, *DEX: Digital evidence provenance supporting reproducibility and comparison*, (2009).

[11] Brothers, S., *How Cell Phone Forensic Tools Actually Work - Proposed Leveling System*, 2009, Mobile Forensics World 2009. Chicago, Illinois.

[12] Brian Carrier, *Open Source Digital Forensics Tools: The Legal Argument*, (2002).

[13] Chet Hosmer, *Chapter 1: Why Python Forensics?*, Python Forensics, Boston, 2014, pp. 2 – 9.

[14] Dave Kuhlman, *Generate Data Structures from XML Schema*, https://pythonhosted.org/generateDS/, Última visita: 5 de setiembre de 2015.

[15] Dianne Hackborn, *READ_LOGS permission not granted*, https://groups.google.com/forum/#!msg/android-developers/6U4A5irWang/AvZsrTdfICIJ, Última visita: 5 de setiembre de 2015.

[16] Eoghan Casey, *Digital Evidence and Computer Crime, Chapter 20*, (2015).

[17] Gartner, *Smartphone Sales Surpassed One Billion Units in 2014*, http://www.gartner.com/newsroom/id/2996817, Última visita: 5 de setiembre de 2015.

[18] Genymobile, *Genymotion: A faster Android Emulator*, https://www.genymotion.com, Última visita: 5 de setiembre de 2015.

[19] Google, *Listado de dispositivos compatibles*, https://support.google.com/googleplay/answer/1727131, Última visita: 5 de setiembre de 2015.

[20] ———, *Project Ara*, http://www.projectara.com/, Última visita: 5 de setiembre de 2015.

[21] Jeff Gamet, *Proposed EU Law Could Force Apple to Change iPhone Charging Connector*, http://www.macobserver.com/tmo/article/proposed-eu-law-could-force-apple-to-change-iphone-charging-connector, Última visita: 5 de setiembre de 2015.

[22] Juan Andrés Diana, José Ignacio Varela, *Recolección de evidencia digital sobre dispositivos móviles - Estado del Arte*, (2015).

[23] Mari DeGrazia, *Finding and reverse engineering deleted SMS messages*, http://az4n6.blogspot.com/2013/02/finding-and-reverse-engineering-deleted_1865.html, Última visita: 5 de setiembre de 2015.

[24] MITRE, *A Python library for parsing, manipulating, and generating CybOX content*, https://github.com/CybOXProject/python-cybox, Última visita: 5 de setiembre de 2015.

[25] ———, *Custom Objects*, http://cybox.readthedocs.org/en/latest/examples.html#custom-objects, Última visita: 5 de setiembre de 2015.

[26] ———, *Cyber Observable eXpression*, http://cybox.mitre.org/documents/Cyber%20Observable%20eXpression%20(CybOX)%20Foundations%20-%20(SwA%20Forum%20Spring%202012)%20-%20Sean%20Barnum.pdf, Última visita: 5 de setiembre de 2015.

[27] ———, *Listado de objetos CybOX v2.1*, http://cyboxproject.github.io/releases/2.1/#cybox-object-downloads, Última visita: 5 de setiembre de 2015.

[28] ———, *Repositorio GitHub de CybOX*, http://cyboxproject.github.io/, Última visita: 5 de setiembre de 2015.

[29] ———, *STIX-TO-HTML*, https://github.com/STIXProject/stix-to-html, Última visita: 5 de setiembre de 2015.

[30] Cindy Murphy, *Cellular Phone Evidence: Data Extraction and Documentation*, (2010).

[31] Namheun Son, et al., *A study of user data integrity during acquisition of Android devices*, (2013).

[32] Nikolay Elenkov, *Unpacking Android backups*, http://nelenkov.blogspot.com/2012/06/unpacking-android-backups.html, Última visita: 5 de setiembre de 2015.

[33] OASIS, *OASIS Advances Automated Cyber Threat Intelligence Sharing with STIX, TAXII, CybOX*, https://www.oasis-open.org/news/pr/oasis-advances-automated-cyber-threat-

intelligence-sharing-with-stix-taxii-cybox, Última visita: 5 de setiembre de 2015.

[34] OpenSignal, *Android Fragmentation Visualized (August 2015)*, http://opensignal.com/reports/2015/08/android-fragmentation/, Última visita: 5 de setiembre de 2015.

[35] Oracle, *JAXB project*, https://jaxb.java.net/, Última visita: 5 de setiembre de 2015.

[36] Oxygen Forensic, *Demo backups*, http://www.oxygen-forensic.com/en/download/devicebackups, Última visita: 5 de setiembre de 2015.

[37] Python Docs, *Built-in Functions: Class Property*, https://docs.python.org/2/library/functions.html, Última visita: 5 de setiembre de 2015.

[38] _____, *importlib: The implementation of import*, https://docs.python.org/3/library/importlib.html, Última visita: 5 de setiembre de 2015.

[39] Timothy Vidas, et al., *Towards a General Collection Methodology for Android Devices*, (2011).

[40] U.S. Department of Justice, *Electronic Crime Scene Investigation: A Guide For First Responders*, (2001).

[41] W. Alink, *XIRAF - XML-based indexing and querying for digital forensics*, (2006).

[42] Wikipedia, *Android version history*, https://en.wikipedia.org/wiki/Android_version_history, Última visita: 5 de setiembre de 2015.

[43] _____, *History of Microsoft Windows versions*, https://en.wikipedia.org/wiki/History_of_Microsoft_Windows, Última visita: 5 de setiembre de 2015.

[44] _____, *iOS version history*, https://en.wikipedia.org/wiki/IOS_version_history, Última visita: 5 de setiembre de 2015.

Apéndice A

Guía de extensión de la herramienta

Una vez determinada la necesidad de contar ya sea con una nueva operación, tipo de dato o tipo de fuente de datos, es importante que tengamos en cuenta los requerimientos que impone la herramienta Android Inspector sobre cada uno de ellos, así como también una serie de buenas prácticas que nos pueden ayudar a desarrollar las mismas con mayor facilidad.

Mediante esta guía pretendemos ilustrar de principio a fin el proceso de desarrollo de cada una de las extensiones que permite realizar la herramienta. Para ello, dividiremos estos procesos en una serie de pasos.

A.1. Desarrollo de una nueva operación

Paso 1: Determinar las propiedades de la operación

Como vimos en la sección 3.4, una operación consiste de un mecanismo que nos permite extraer información de un cierto tipo de dato desde una fuente de datos, soportando un cierto conjunto de modelos de dispositivos y versiones de Android. Esta es la información más básica con la que podemos contar de una operación.

Usualmente cuando ya hemos decidido que precisamos desarrollar una nueva operación, tenemos una idea de esta información básica. Veamos cómo cada

uno de estos datos determinan diversos aspectos de la operación.

Propiedad	Descripción
Tipo de dato	*Determina el tipo de objeto CybOX utilizado para representar los datos.*
Fuente de datos	*Determina el extractor y los parámetros utilizados por la operación.*
Versiones de Android	*Determina las versiones del sistema operativo Android soportadas por la operación.*
Modelos de dispositivos	*Determina los modelos de dispositivos Android soportados por la operación.*

Cuadro A.1: Propiedades básicas de una operación

Para estas dos últimas propiedades, si bien uno generalmente parte con el objetivo de soportar un determinado conjunto de dispositivos y versiones de Android, luego de desarrollada la operación podemos descubrir que la misma funciona correctamente en un rango más amplio de dispositivos y versiones de Android del que fue inicialmente verificada. Por lo tanto, podemos reflejar esta información utilizando estas propiedades.

Paso 2: Evaluar componentes a reutilizar

Uno de los aspectos más interesantes que propone el diseño de la herramienta es facilitar la reutilización de componentes. En particular, estamos hablamos de *extractors* y *cybox objects*. De esta forma, debemos evaluar si no contamos ya con componentes que nos permitan extraer información de la fuente de datos deseada ó que nos permitan representar el tipo de dato que buscamos.

En el caso de *extractors*, recordemos que los mismos nos permiten obtener datos de cierto tipo de fuente de datos. Por lo tanto, debemos evaluar si la *fuente de datos* de la cual pretendemos extraer datos se encuentra comprendida por alguno de los *tipos de fuente de datos* con los que contamos.

A modo de ejemplo, supongamos que deseamos extraer datos de la aplicación para Android de Twitter. En este caso, podríamos utilizar el extractor que brinda por defecto *Android Inspector* para obtener datos de aplicaciones, esto es, el *ApplicationExtractor*. Para indicarle la aplicación de la cual deseamos extraer los datos, le pasamos como valor del parámetro *package_name*

el nombre correspondiente al paquete de la aplicación, que en este caso es *com.twitter.android*.

Si tras evaluar los extractors con los que contamos vemos que ninguno satisface nuestros requerimientos, tendremos que considerar desarrollar un nuevo extractor, y en consecuencia definir un nuevo tipo de fuente de datos. La sección A.3 describe los pasos a tomar para esto.

En el caso de *cybox objects*, el propio lenguaje CybOX nos ofrece un conjunto de objetos ya definidos para representar diversos tipos de datos. El listado de objetos definidos por CybOX se encuentra en [27].

Si tras revisar los objetos definidos por el lenguaje CybOX vemos que el tipo de dato que buscamos representar no se encuentra definido o deseamos agregar campos a un objeto existente, tenemos la posibilidad de diseñar lo que CybOX denomina un *custom cybox object* para representar un nuevo tipo de dato. Para ello, la sección A.2 describe los pasos a tomar y aspectos que deberíamos tener en cuenta.

Paso 3: Implementar el inspector

Llegados a este punto, tenemos claro lo que debe realizar la operación y contamos tanto con el extractor adecuado para obtener los datos como con el objeto CybOX adecuado para representar la información que buscamos. Nos resta implementar el procedimiento mediante el cual se realizará la examinación de los datos extraídos, esto es, el *inspector*.

Antes de entrar en los detalles de implementación de un inspector, es oportuno que mencionemos un detalle importante. Tanto los *inspectors* como los *extractors* y los *cybox objects* (que veremos cómo implementarlos en las próximas secciones) que maneja la herramienta son implementados como módulos de extensión en el lenguaje Python. Ya vimos en la sección 5.1, cómo Python es un lenguaje ideal en el cual realizar este tipo de extensiones.

Tengamos en cuenta que para el desarrollo de estos módulos podemos utilizar cualquiera de las bibliotecas estándar que brinda Python, además de la biblioteca python-cybox vista en la sección 5.3 para facilitar el manejo y producción de contenido CybOX. Por ejemplo, para examinar bases de datos SQLite contamos con la biblioteca estándar sqlite3. Además de estos módulos, en la sección A.4 veremos un par de módulos utilitarios que incluímos en la herramienta para facilitar el desarrollo en base a la experiencia adquirida

desarrollando los diversos módulos del caso de estudio visto en el Capítulo 6.

Vista la plataforma que contamos para desarrollar, ahora veremos cómo implementar el inspector de la operación.

Lo primero que debemos realizar es crear un nuevo módulo Python. El mismo deberá contener una clase que implemente la interfaz *Inspector* (introducida en la sección 5.4):

```
class Inspector(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def execute(self, device_info, extracted_data_dir_path):
        """
        :type device_info: DeviceInfo
        :type extracted_data_dir_path: string
        :rtype : (list(Object), list(FileObject))
        """
        pass
```

Nota: El nombre del módulo del inspector debe ser expresado en *Snake Case* mientras que el nombre de la clase que implementa la interfaz debe ser expresado en *Camel Case*. Ambos nombres deben coincidir tras realizar la conversión de notación. Esto es, si por ejemplo el módulo utiliza el nombre *contact_facebook_inspector*, la clase que implementa la interfaz Inspector deberá llamarse *ContactFacebookInspector*.

A continuación veremos varios aspectos que debemos tener en cuenta al implementar un inspector.

Empecemos por observar los parámetros del método `execute` que ofrece la interfaz *Inspector*

- *device_info*: Nos brinda información sobre el dispositivo (modelo y versión de Android) del cual se realizó la extracción. Muy a menudo resulta útil conocer estos datos debido a que ciertos aspectos son implementados de diferente forma dependiendo del fabricante o versión de Android siendo utilizada. De esta forma, tenemos la posibilidad de decidir cómo realizar la examinación de los datos en función de esta información.
- *extracted_data_dir_path*: Nos indica la ruta al directorio en donde se encuentran los archivos obtenidos de la fuente de datos designada.

En cuanto a las salidas, la interfaz Inspector establece que el método execute

debe retornar:

- *Una lista de Object*, representando la información encontrada correspondiente al tipo de dato que maneja la operación.
- *Una lista de FileObject*, representando la información sobre los archivos originales de los cuales se obtuvo la información.

La primera lista debe contener objetos CybOX de un único tipo (SMSMessageObject, por ejemplo). Además, cada uno de dichos objetos debe referenciar a los objetos (que se encuentran en la segunda lista) que representan a los archivos de los cuales se obtuvo la información expresada. Para ello, indicamos dicha relación de la siguiente forma:

```
inspected_object.add_related(source_object,
                             ObjectRelationship.TERM_EXTRACTED_FROM,
                             inline=False)
```

En la segunda lista vimos que debemos representar la información de cada uno de los archivos originales de los cuales obtuvimos información. Para esto, podemos utilizar el método `create_file_object` del módulo utilitario *inspectors_helper.py* indicándole la ruta en donde se encuentra el archivo extraído y la ruta original en donde se encontraba en el dispositivo:

```
file_object = inspectors_helper.create_file_object(file_path,
                                                   original_file_path)
```

En cuanto al manejo de errores, la clase que implemente el inspector debe lanzar la excepción *OperationError* indicando un mensaje apropiado a fin de informar al usuario en los debidos casos que pueda ocurrir una falla en el proceso de examinación.

Por otro lado, cabe mencionar que no es necesario implementar toda la lógica del inspector en una única clase sino que podemos dividirla en varias clases dentro del módulo que estamos desarrollando.

Finalmente, con el fin de facilitar el debugging de inspectors, incluímos una bandera denominada `simple_output` que elimina los namespaces y utiliza enteros simples en vez de UUIDs en todos los identificadores de contenido CybOX que maneja el inspector. Mediante este mecanismo podemos obtener salidas de contenido CybOX mucho más legibles en XML a los fines de verificar el correcto funcionamiento del inspector siendo desarrollado. Para

activar esta modalidad simplemente debemos inicializar Android Inspector indicándole la bandera mencionada de la siguiente forma:

```
python andi.py --simple-output
```

Paso 4: Definir la operación

Finalmente, ahora que tenemos todos los componentes de la operación implementados, estamos en condiciones de escribir su archivo de definición. El mismo permitirá a un usuario de la herramienta importar de forma sencilla la operación en su ambiente utilizando el comando `add_ext --type operation`. Para esto, debemos escribir su definición en formato JSON y empaquetarla junto al módulo del inspector implementado en la sección anterior en un archivo tar.

El **archivo de definición** de una operación debe tener el siguiente formato:

```
{
   "name": <string>,
   "data_type": <string>,
   "data_source_type": <string>,
   "data_source_param_values": <dict(string, string)>,
   "inspector_name": <string>,
   "android_versions": <array(string)>,
   "device_models": <array(string)>
}
```

A continuación veremos los detalles de cada uno de los campos de la definición:

Nombre	Descripción
name	Nombre de la operación. Tiene el formato *namespace:name*, de forma de ser un nombre único.
data_type	Nombre del tipo de dato utilizado.
data_source_type	Nombre del tipo de fuente de dato utilizada.

data_source_param_values	Contiene los valores correspondientes a los parámetros requeridos por el tipo de fuente de datos utilizada.
inspector_name	Nombre de la clase que implementa el inspector de la operación.
android_versions	Versiones de Android que soporta la operación. Los identificadores de versiones de Android pueden ser consultados en [9].
device_models	Modelos de dispositivos que soporta la operación. Los identificadores de modelos de dispositivos pueden ser consultados en [19].

Cuadro A.2: Campos de la definición de una operación

A.2. Desarrollo de un nuevo tipo de datos

Paso 1: Diseñar e implementar el nuevo objeto CybOX

Al pensar en crear un nuevo tipo de datos, primero debemos diseñar un nuevo objeto CybOX para representarlo y luego implementar un módulo Python para permitir al desarrollador de un inspector trabajar con el mismo de igual forma que con los objetos CybOX provistos por la biblioteca *python-cybox*.

La sección 2.4.1.6 detalla cómo proceder para diseñar un nuevo objeto CybOX. Ahora veamos cómo debemos proceder para implementar un nuevo objeto CybOX.

En primer lugar, debemos crear una nueva clase que represente al tipo de dato que estamos desarrollando. De esta forma, la misma debe heredar de la clase *Custom* provista por *python-cybox* y exponer los campos del nuevo objeto. La forma de exponer estos campos debe ser mediante atributos públicos (para ser coherentes a cómo son expuestos los campos por parte de los objetos CybOX que provee la biblioteca).

Para exponer los campos del objeto como mencionamos, debemos de tener en cuenta que la clase Custom utiliza una colección de *CustomProperty* mediante la cual se definen los campos del objeto en runtime. Para lograr contar con una interfaz igual a la provista por los demás objetos CybOX, esto es, poder acceder a los atributos de la clase directamente, podemos utilizar la construcción de *Property* que brinda Python [37]. De esta forma, podemos

interceptar el momento en que se asigna y accede al atributo, y actuar como proxy para utilizar la *CustomProperty* correspondiente.

Nota: La nueva clase debe ser escrita en un nuevo módulo Python de forma que siga la misma convención de nombres descrita anteriormente para un inspector.

Paso 2: Definir el tipo de dato

Para que el usuario pueda importar fácilmente el nuevo tipo de dato utilizando el comando `add_ext --type data_type`, debemos crear su definición en formato JSON y empaquetarla en un archivo tar junto al módulo Python implementado en la sección anterior.

El **archivo de definición** de un tipo de dato debe tener el siguiente formato:

```
{
   "name": <string>,
   "cybox_object_name": <string>
}
```

A continuación veremos los detalles de los campos de la definición:

Nombre	Descripción
name	Nombre del tipo de dato. Al igual que operation, tiene el formato *namespace:name*.
cybox_object_name	Nombre de la clase del objeto CybOX que representa al tipo de dato.

Cuadro A.3: Campos de la definición de un tipo de dato

A.3. Desarrollo de un nuevo tipo de fuente de datos

Paso 1: Implementar el extractor

Como vimos en la sección 4.6, el extractor implementa el mecanismo que se encarga de obtener los datos de un cierto tipo de fuente de datos. Por lo tanto, es lo primero que debemos considerar.

Veremos que varios de los aspectos a tener en cuenta son similares a los vistos con el inspector. De igual forma que en el otro caso, comenzamos creando un nuevo módulo Python que contenga una clase que implemente la interfaz *Extractor* (introducida en la sección 5.4):

```
class Extractor(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def execute(self, extracted_data_dir_path, param_values):
        """
        :type extracted_data_dir_path: string
        :type param_values: dict(string, string)
        :rtype : None
        """
        pass
```

Nota: Los nombres del módulo extractor y la clase que implementa su respectiva interfaz deben cumplir con las mismas reglas descritas anteriormente para el inspector.

A continuación, consideraremos los aspectos que debemos tener en cuenta al implementar un extractor.

Observando la interfaz *Extractor*, vemos que el método `execute` nos proporciona los siguientes parámetros:

- *extracted_data_dir_path*: Indica la ruta al directorio en el cual el extractor debe almacenar todos los datos extraídos en esta etapa.
- *param_values*: Indica los valores de los parámetros requeridos por el extractor. Cada fuente de datos de un mismo tipo establece diferentes valores para estos parámetros. Por ejemplo, en el caso del *ApplicationExtractor*, el parámetro *package_name* tendrá diferentes valores dependiendo de la aplicación que tomemos como fuente de datos.

Como salida, si bien el método `execute` no retorna un resultado, ya mencionamos que se espera que el extractor almacene los datos extraídos en la ruta que se le especificó del filesystem de la maquina *host*. Por otro lado, la forma en que el extractor almacena los datos en dicha ruta es un aspecto importante que se debe tener en cuenta. El desarrollador del extractor debe establecer una clara especificación de cómo el extractor almacena los datos extraídos de forma que luego todos los inspectors de las operaciones que utilizan el extractor sepan acceder a los datos extraídos.

En el caso de los extractors que desarrollamos nosotros en este trabajo (*ApplicationExtractor* y *AdbBackupExtractor*), ambos se dedican a extraer los datos de una aplicación y ambos los almacenan utilizando la misma especificación. Este último detalle no es un requisito que debamos cumplir al implementar un extractor, pero se realizó a propósito de forma que las operaciones que examinan datos de aplicaciones pudieran intercambiar fácilmente el extractor utilizado. De esta forma, vemos la importancia que puede tener especificar cómo se almacenan los datos extraídos.

En cuanto al manejo de errores, debemos utilizar la excepción *OperationError*, de forma similar a como lo realizamos en un inspector, acompañando la misma de un mensaje que describa el error ocurrido.

Por último, cabe mencionar que la herramienta no impone ningún tipo de restricción en cuanto a la forma utilizada para realizar la extracción de los datos ni el medio a través del cual se debe realizar la extracción. El único requisito que impone es que la misma debe almacenar los datos extraídos en el directorio que se le especifica. Dicho esto, el desarrollador puede optar por utilizar el protocolo adb, que brinda la ventaja de tener un acceso universal a todos los dispositivos ya sean reales o virtuales, ó bien puede optar por utilizar otro medio.

Paso 2: Definir el tipo de fuente de datos

De forma similar que para las extensiones anteriores, debemos crear una definición para el tipo de fuente de dato en formato JSON y empaquetarla junto al módulo del extractor implementado en un archivo tar. De esta forma, el usuario podrá importar el nuevo tipo de fuente de datos utilizando el comando `add_ext --type data_source_type`.

El **archivo de definición** de un tipo de fuente de datos utiliza JSON y tiene el siguiente formato:

```
{
        "name": <string>,
        "extractor_name": <string>,
        "required_params": <array(string)>
}
```

A continuación veremos los detalles de cada uno de los campos de la definición:

Nombre	Descripción
name	Nombre del tipo de fuente de dato. Al igual que operation, tiene el formato *namespace:name*.
extractor_name	Nombre del extractor responsable de obtener los datos de este tipo de fuente de datos.
required_params	Listado de parámetros requeridos por este el extractor.

Cuadro A.4: Campos de la definición de un tipo de fuente de datos

A.4. Módulos utilitarios

La herramienta cuenta con un paquete que provee módulos que brindan soluciones a tareas que son frecuentemente requeridas al desarrollar extensiones, tanto para extractors como inspectors. De esta forma, buscamos facilitar el desarrollo de las mismas poniendo a disposición código que pueda ser reutilizado. Concretamente contamos con los siguientes dos módulos:

1. *inspectors_helper.py*: Este módulo brinda varias facilidades para el desarrollo de un inspector:
   - `create_file_object`: Crea el File Object con todos los datos relevantes (nombre, extensión, ruta, formato, tamaño y hash) a partir de la ruta al archivo y la ruta en donde se encontraba almacenado originalmente.
   - `execute_query`: Ejecuta una consulta en la base de datos indicada por parámetro, devolviendo el resultado en un cursor conteniendo las filas resultantes y el nombre de sus columnas correspondientes.

- `get_app_version_name`: A partir de un ruta a un APK, devuelve el nombre de versión de la aplicación. Para esto utiliza el comando `aapt` para extraer el archivo AndroidManifest compilado y luego parsearlo en búsqueda del dato.

2. *adb.py*: Este módulo es un wrapper de la herramienta adb. El mismo fue desarrollado por Google para testear esta herramienta en AOSP [6]. Usando este wrapper se facilita la interacción con la herramienta de línea de comando ADB.

Apéndice B

Diagramas de flujo de los comandos

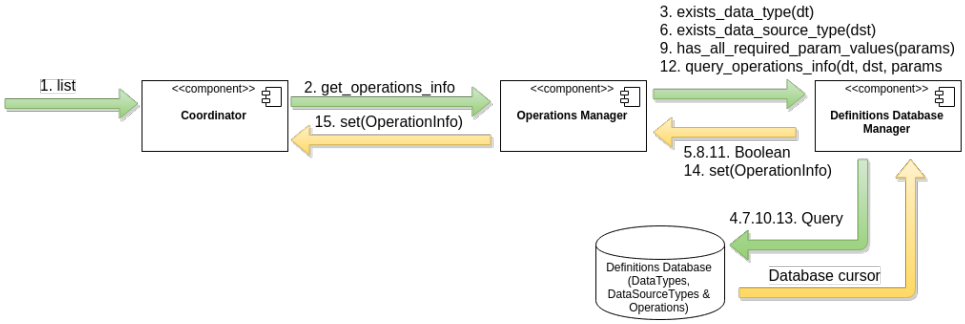

Figura B.1: Flujo de ejecución del comando `list`

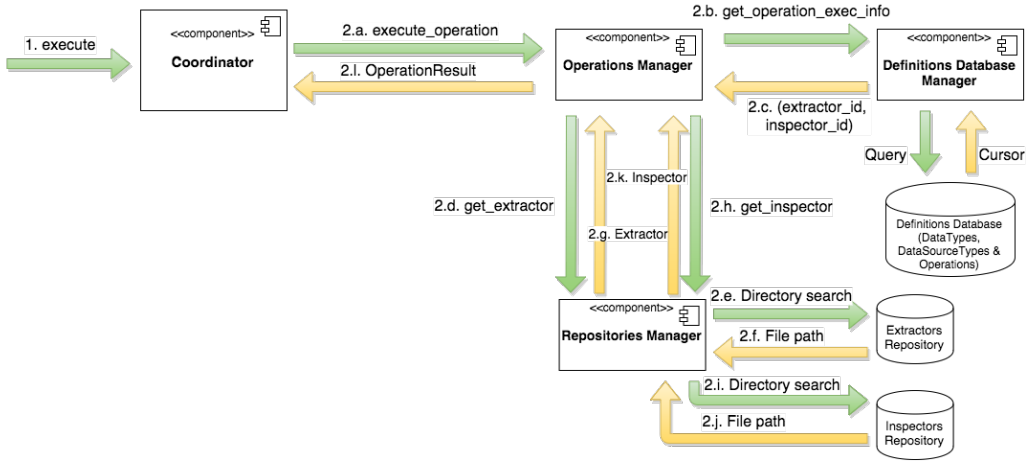

Figura B.2: Flujo de ejecución del comando `execute`

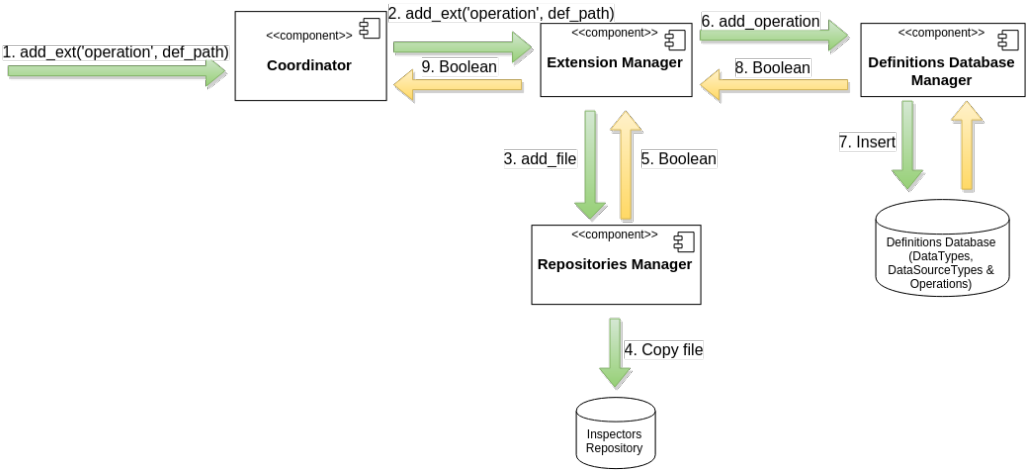

Figura B.3: Flujo de ejecución del comando `add_ext` para el caso de adición una operación

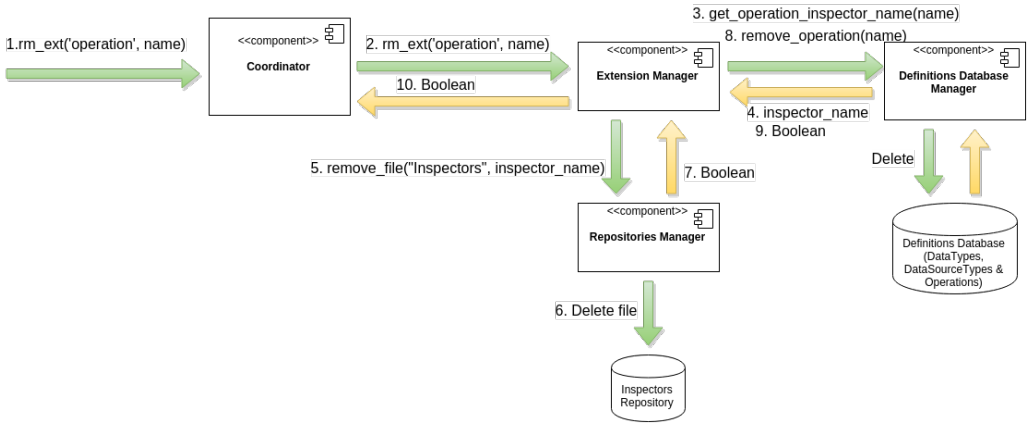

Figura B.4: Flujo de ejecución del comando `rm_ext` para el caso de eliminación una operación

Recolección de evidencia digital sobre dispositivos móviles

Estado del arte

Autores
Juan Andrés Diana Robledo
José Ignacio Varela Gallino

Supervisores
Gustavo Betarte
Marcelo Rodríguez

Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
2015

Índice general

Capítulo 1

Características de los dispositivos móviles

Empezaremos por conocer los aspectos fundamentales de este tipo de dispositivos que serán nuestro objeto de estudio. Entender sus características particulares es de suma importancia si pretendemos conocer cómo extraer información de los mismos y entender qué consecuencias tienen nuestras acciones a lo largo de este proceso.

En el área forense, contamos con un desarrollo del conocimiento sobre computadoras personales (PC) muy avanzado. El mismo ha sido posible debido a que si bien en las últimas décadas las capacidades de las PC han estado en constante incremento, las bases de su funcionamiento e interfaces brindadas no han presentado cambios radicales. Esto ha permitido que hoy en día en las PC, al utilizar determinadas metodologías y procedimientos, podamos contar con ciertas garantías sobre los datos extraídos que resultan muy importantes del punto de vista forense.

Las PC son los sistemas informáticos que presentan una mayor semejanza a los dispositivos móviles de hoy en día, debido a la gran cantidad de aspectos que comparten. Por lo tanto, resulta razonable que intentemos aplicar el conocimiento con el que contamos en el área forense sobre las PC, también a los dispositivos móviles. Esto es posible en ciertos aspectos en los que comparten las mismas características con las PC. Sin embargo, veremos que los dispositivos móviles presentan una serie de características muy diferentes, las cuales implican que a menudo no sea posible aplicar las mismas técnicas y debamos buscar nuevas alternativas.

Además de introducir nuevos desafíos desde el punto de vista forense, los dispositivos móviles también brindan nuevas oportunidades al contar con ciertos tipos de información a los cuales no estábamos acostumbrados a encontrar en las PC.

1.1. Aspectos generales

Antes de adentrarnos en las características particulares de los dispositivos móviles, veamos primero una forma de clasificar a los teléfonos móviles que toma en cuenta las características en conjunto que brinda un dispositivo y los divide en: feature phones y smartphones. Esta clasificación nos permitirá tener una idea del tipo de información que esperamos encontrar en un dispositivo.

Los denominados feature phones son dispositivos destinados primordialmente al intercambio de voz y mensajes, mientras que los smartphones ofrecen capacidades más avanzadas comparables a las ofrecidas por una PC. Los mismos ofrecen soporte de voz, mensajes de texto y un conjunto de aplicaciones básicas relacionadas a Personal Information Management (PIM) como son la agenda de contactos y el calendario [34].

Los smartphones, además de contar con las funcionalidades básicas que ofrecen los anteriores, añaden la capacidad de correr una amplia variedad de aplicaciones. Las mismas generalmente son distribuidas de forma centralizada a través de una tienda virtual.

Tanto si los comparamos desde el punto de vista de su hardware como de su software, los smartphones suelen ser mucho más sofisticados. Del lado del hardware, éstos usualmente poseen procesadores con múltiples núcleos y pantallas más grandes, de altas densidades de píxeles y con capacidades táctiles. Del lado del software, utilizan sistemas operativos propietarios u open source y proveen SDKs con amplia documentación para el desarrollo de aplicaciones. La totalidad del mercado de estos sistemas es cubierta por un conjunto muy reducido de opciones: Android, Blackberry OS, Firefox OS, iOS, Symbian y Windows Phone [18]. En los feature phones, en cambio, podemos encontrar una inmensa cantidad de sistemas operativos cerrados, los cuales suelen carecer de documentación pública [34].

Finalmente, es importante tener un panorama de la situación en la que nos encontramos: El número de smartphones vendidos a nivel mundial superó por

primera vez en la historia al de feature phones en el último cuarto de 2013, según un reporte realizado por Gartner [18]. Resulta clave tener en cuenta esto para invertir en conocimiento al largo plazo pero también nos recuerda que si bien los smartphones son el futuro, la cantidad de feature phones sigue siendo sumamente relevante en el mundo móvil.

1.2. Aspectos de hardware

La cantidad de modelos de dispositivos móviles que existen y su variedad están en continuo aumento. Si consideramos únicamente aquellos dispositivos móviles que corren el sistema operativo Android (el más popular a nivel mundial en este momento [18]), podemos observar que a julio de 2013 existían más de 10,000 modelos distintos [36].

¿Por qué resulta relevante considerar la inmensa cantidad de modelos de dispositivos móviles que existen, si después de todo, podemos observar que en el terreno de las PC también existe una gran variedad de modelos y sin embargo las herramientas forenses con las que contamos hoy en día no presentan mayores dificultades en soportar esta diversidad?

Resulta que los dispositivos móviles presentan, en la actualidad, una particularidad muy importante que hace que la gran diversidad de modelos sea un hecho relevante. Se trata de que presentan una gran variedad de interfaces de hardware, incluso entre modelos de un mismo fabricante. Esto se ve acentuado por el hecho de que muy a menudo los dispositivos móviles cuentan con hardware especializado. Esta diversidad de interfaces se extiende y probablemente es más visible al público en general al observar la cantidad de conectores de cables de poder y datos que existen para dispositivos móviles. La Unión Europea incluso intentó impulsar regulaciones con el objetivo de obligar a los fabricantes de estos dispositivos a adoptar un estándar único para los conectores de teléfonos celulares [41].

La diferencia a apreciar en la industria de las PC es que el hardware es diseñado con una visión modular para proveer compatibilidad con otras piezas de hardware construidas potencialmente por otros fabricantes. Por ejemplo, en caso de querer acceder a los datos de un disco duro de un PC, contamos con un conjunto de interfaces estándar muy limitado en la industria (IDE, SATA, SCSI).

Si consideramos ahora a los dispositivos móviles, vemos que proveer inter-

operabilidad entre componentes de hardware de diversos fabricantes no es un objetivo que suelen tener en mente los fabricantes a la hora de diseñar un nuevo dispositivo. Frecuentemente priorizan el poder contar con un diseño compacto, muchas veces soldando los componentes directamente o diseñando chips que integran varias funciones con el objetivo de disminuir el espacio. Esto se evidencia en las arquitecturas SoC (System on a Chip), populares en el mundo móvil. Muchos dispositivos hoy en día vienen incluso con su batería soldada al PCB por esta misma razón.

Mientras en la industria la tendencia de dispositivos móviles con diseños monolíticos se acentúa, un grupo de investigación de Google (Advanced Technology and Projects group) ha tenido la iniciativa, en estos últimos años, de formar la base para contar con dispositivos móviles modulares. Bajo el nombre de 'Project Ara', proponen una plataforma que brinde el esqueleto mínimo necesario de interfaces estándar con el objetivo de permitir que fabricantes de diversos tipos de piezas de hardware de dispositivos móviles (como cámara, radio, batería, almacenamiento, etc) puedan diseñarlas de forma que interoperen con facilidad con componentes de otros fabricantes. El usuario del dispositivo entonces tendría la libertad de escoger componentes de diferentes fabricantes según sus necesidades y armar su dispositivo a medida. El proyecto se encuentra en una etapa temprana con prototipos avanzados e interés de varios fabricantes [19].

Es interesante contemplar por un instante esta iniciativa ya que su éxito podría tener una repercusión significativa en el proceso de adquisición de datos. El hecho de contar con interfaces estándar de hardware podría facilitar en gran medida el proceso. Además, brindaría la habilidad de poder remover físicamente los módulos de hardware con facilidad. Esta última es una diferencia notoria que presentan hoy en día los dispositivos móviles. En particular, cuando consideramos el acceso a datos almacenados, a diferencia de las PC, la unidad de almacenamiento principal se encuentra soldada al PCB del dispositivo. La única forma de remover dicha unidad es mediante técnicas de "chip off" como veremos más adelante en la sección de "Tipos de métodos de extracción". La excepción a esto son las tarjetas de expansión (SD cards), las cuales se encuentran presentes en muchos modelos. Veremos más adelante que si bien son una fuente de información importante, los datos más sensibles suelen estar en el almacenamiento principal.

1.2.1. Interfaces de entrada/salida

Estas son esenciales a considerar dado que son los puntos a través de los cuales el dispositivo interactúa con el mundo exterior, recibiendo y enviando información. Mientras que las interfaces de entrada determinarán en gran medida los tipos de datos que podremos encontrar en un dispositivo, las interfaces de salida nos impondrán los medios disponibles que tendremos para extraer los datos.

Como interfaces de entrada, los dispositivos móviles suelen contar con un conjunto básico de botones (encendido, volumen, etc). Dependiendo del dispositivo, la interfaz principal de entrada brindada al usuario puede ser un teclado o una pantalla táctil. Además, los mismos cuentan (y cada vez en mayor cantidad) con una diversidad de sensores para capturar datos del entorno. Estos sensores van desde los más comunes como micrófono, cámara fotográfica, GPS y giroscopio hasta otros más novedosos como cámara de profundidad, sensor de huellas dactilares, NFC, etc. Esta mayor cantidad y amplia gama de sensores que poseen los dispositivos móviles marca una gran diferencia con las PC. Nos brindan nuevos tipos de datos con los que podemos contar. Además del valor que nos puedan otorgar estos datos por sí mismos, resulta interesante considerar el valor que pueden brindar al realizar posteriormente un análisis en conjunto de los mismos, pudiendo brindar una nueva dimensión a la información.

En cuanto a las interfaces de salida, la interfaz principal que suelen brindar los dispositivos móviles para interactuar con el usuario es una pantalla gráfica. Adicionalmente, es común también contar con parlantes y un módulo de vibración, en especial con el fin de tener una forma de comunicarse con el usuario cuando éste no está prestando atención a la pantalla. Otra interfaz que provee la gran mayoría, incluso los feature phones, es un puerto de comunicación (como puede ser USB) mediante el cual permite al usuario exportar ciertos datos del dispositivo (como pueden ser datos de contactos, calendario, mensajes o archivos multimedia) al conectarlo a un PC. Para ello, el fabricante del dispositivo usualmente provee una aplicación cliente la cual ejecuta en la PC y mediante un protocolo realiza operaciones de extracción de datos. Usualmente los datos que brindan estas aplicaciones son muy limitados.

En la sección de "Tipos de métodos de extracción"veremos las implicaciones que tienen las distintas interfaces de entrada y salida al ser utilizadas para extraer datos.

1.2.2. SIM Cards

Este es el nombre por el que se conocen popularmente las pequeñas smart cards utilizadas por los dispositivos móviles para interoperar con las redes celulares.

Si consideramos el modelo GSM [34], una estación móvil consta de dos componentes:

1. La **UICC** (Universal Integrated Circuit Card), esto es, un componente removible el cual contiene información esencial sobre el suscriptor.

2. El **dispositivo móvil**, el cual posee un módulo de radio. Sin embargo, el mismo no es capaz de brindar todas sus funcionalidades sin una UICC presente.

El propósito principal de la UICC es autenticar al usuario del dispositivo móvil de forma segura ante el proveedor de red celular y brindarle acceso a aquellos servicios al cual se encuentra suscripto. Adicionalmente, ofrece cierta capacidad bastante limitada para almacenar información personal del usuario como son entradas de agenda telefónica, mensajes de texto, últimos números marcados (LND) e información relacionada a servicios.

La partición que establece el modelo GSM trae cierto grado de portabilidad. Le permite al suscriptor trasladar su identidad, información asociada y acceso a servicios de forma automática entre dispositivos móviles compatibles dentro de una red GSM. Esto no suele ser así en dispositivos móviles CMDA ya que los mismos generalmente no cuentan con una UICC e integran directamente al dispositivo móvil dicha funcionalidad. Sin embargo, los nuevos modelos (4G/LTE) de dispositivos CMDA ahora también utilizan UICCs y brindan esta portabilidad.

Originalmente, cuando surgió GSM la UICC únicamente corría una aplicación, denominada SIM (Subscriber Identity Module). Es por esto que frecuentemente se utiliza el término SIM como sinónimo de la tarjeta física en lugar de UICC. Sin embargo, hoy en día existen otras aplicaciones que pueden correr en la UICC, incluso de forma simultánea. Por ejemplo, para redes UMTS se utiliza una aplicación denominada Universal Subscriber Module (USIM), mientras que para redes CMDA 4G se utiliza otra llamada CDMA Subscriber Identity Module (CSIM).

La UICC es un tipo especial de smart card. La misma está disponible en

varios tamaños: Mini SIM, Micro SIM o Nano SIM. Además, cuenta con un identificador único denominado ICC-ID (Integrated Circuit Card Identified) que normalmente va impreso en el frente de la tarjeta física, pero también es almacenado en la propia tarjeta y es posible leerlo.

Las UICC cuentan con un microprocesador, ROM y RAM. Además, poseen su propio sistema operativo y file system en el cual se almacena la información personal y de servicio del usuario. El sistema operativo controla el acceso a los elementos del file system. No toda la información que contiene la UICC puede ser accedida de forma fácil. Acciones como leer o modificar datos pueden ser permitidas o denegadas de forma incondicional, ó de forma condicional bajo la protección de un código PIN. Este código se utiliza para proteger los datos del suscriptor y sólo admite un número muy reducido de intentos (usualmente tres) antes de bloquearse [11].

Por las características vistas, la UICC puede ser considerada un sub-procesador el cual presenta una interfaz al dispositivo. El file system reside en memoria no volátil y está organizado en forma de árbol jerárquico. Por ejemplo, las aplicaciones SIM se componen de tres tipos de elementos: la raíz del file system (MF), directorios de archivos subordinados (DF) y archivos conteniendo datos elementales (EF). Los EF debajo del directorio DF_{GSM} y $DF_{DCS1800}$ contienen información relacionada a las diferentes bandas de frecuencia de operación, y los EF debajo del directorio $DF_{TELECOM}$ contiene información relevante al servicio.

Vemos entonces que las aplicaciones SIMs incluyen información relevante a la red y al usuario. Entre esta información, podemos encontrar tres piezas claves:

- Una clave de autenticación (Ki), necesaria para establecer la conexión con la red.
- Un Personal Identification Number (PIN), para restringir el uso de la tarjeta.
- Un Mobile Subscriber ISDN (MSISDN), este es el número de teléfono del suscriptor.

Usualmente, también se suele encontrar otras piezas de información de interés forense como:

- Un International Mobile Subscriber Identity (IMSI), que identifica de forma única a un suscriptor.

- Un Temporary Mobile Subscriber Identity (TMSI), que se utiliza para evitar revelar el IMSI al transmitir datos por el aire a aquellos que puedan estar interceptando señales de radio.
- Un Location Area Identity (LAI), que identifica geográficamente donde se encuentra una estación móvil.

Tanto el TMSI como el LAI suelen cambiar cada vez que el dispositivo móvil se mueve a una nueva área de la red celular.

1.2.3. Almacenamiento

Acabamos de ver cómo las SIM Cards tienen su propio espacio de almacenamiento. Ahora nos concentraremos en describir las características de los medios de almacenamiento que suelen ser utilizados en dispositivos móviles.

A diferencia de las PC, en las que estamos acostumbrados a encontrar discos magnéticos como medio de almacenamiento, en los dispositivos móviles se utiliza extensamente memoria flash. A continuación, analizaremos las razones que han llevado a esto y luego veremos las diversas formas mediante las que es utilizada esta tecnología en los dispositivos móviles.

1.2.3.1. Memoria Flash

Empecemos por enumerar tres importantes diferencias que brindan los medios de almacenamiento que utilizan memoria flash frente a los discos duros:

- Un menor consumo de energía
- No poseen partes móviles (por lo que son menos propensos a dañarse)
- Menor tamaño físico y más livianos

Estas características hacen que su uso en dispositivos móviles sea ideal al alinearse con varios de los mayores problemas que se priorizan al diseñar un dispositivo móvil.

De igual manera, estas ventajas también son vistas con mucho entusiasmo desde el sector de las PC y con la adición de otras importantes características como mayor velocidades de lectura, escritura y tiempo de acceso, el uso de memoria flash en computadoras de escritorio y en especial laptops ha ido aumentando rápidamente a medida que los precios de las mismas han ido bajando y sus capacidades aumentando.

Veamos ahora qué características interesantes posee esta tecnología desde el punto de vista forense y el impacto que las mismas provocan en la extracción de datos.

La mayor ventaja que ofrece esta tecnología de almacenamiento desde el punto de vista forense es que los dispositivos pueden contener datos borrados aún después de que el individuo haya intentado hacerlos irrecuperables. Esto se debe a las siguientes características que posee la memoria flash [11]:

- Es resistente a impactos físicos, altas temperaturas y presión, por lo que hace difícil su destrucción física exitosa.
- Cuenta con una capacidad limitada de escrituras y sus datos sólo pueden ser borrados por bloque. Esto hace que para borrar un dato generalmente se espere a que el bloque que lo contiene se encuentre lleno antes de realizar el borrado físico del mismo.

Otra característica importante es que los medios de almacenamiento flash es que utilizan algoritmos propietarios para realizar el balanceo de las escrituras a lo largo de todas las celdas de la memoria flash, con el fin de prolongar la vida del medio de almacenamiento. Esto resulta en que los datos borrados puedan permanecer por tiempos considerables antes de que el algoritmo se encargue de borrarlo de forma definitiva.

Como consecuencia de lo anterior, los datos internamente no se almacenan de forma lineal como en los discos duros. Las direcciones físicas de los datos y las expuestas por el controlador al sistema operativo difieren. Esto hace que su funcionamiento sea bastante más complejo comparado al de los discos duros.

Además, la forma de trabajo interna de estos medios de almacenamiento difiere dependiendo del fabricante del controlador debido a que cada uno utiliza algoritmos distintos para realizar el balanceo de escrituras.

El algoritmo de balanceo de escrituras corre de forma periódica cuando el medio de almacenamiento se encuentra operando y no hay posibilidad de deshabilitar el mismo. Esto provoca que desde el punto de vista forense no sea posible realizar una imagen física del medio de almacenamiento con el mismo valor forense como con discos duros, ya que por el simple hecho de estar operativo el medio de almacenamiento, su contenido va a ser modificado [6].

Utilizar un write blocker no es de utilidad como lo sería en el caso de discos duros dado que el proceso de balanceo de escrituras que se da de forma contínua es un proceso interno. De esta forma, debemos ser cuidadosos al trabajar con medios de almacenamientos flash dado que podemos estar borrando datos de valor de forma inadvertida al operar con los mismos.

1.2.3.2. Estándares de tecnologías de almacenamiento Flash

Anteriormente vimos las ventajas que ofrece el almacenamiento en memoria flash y por qué el mismo es el más utilizado en dispositivos móviles. A continuación veremos concretamente cuáles son los diversos "form factors.[en] los cuales podemos encontrar a esta tecnología de almacenamiento.

1. **SSD (Solid State Disk)**: Utilizan un controlador y chips de memoria (NAND generalmente) para persistir datos de forma permanente. Se caracterizan por brindar interfaces de entrada/salida compatibles con las de disco duro, como puede ser SATA. Esto facilita el reemplazo de los últimos. Además, es común que su formato de empaquetado sea el típico formato de 2,5 pulgadas de los disco duros. Por dichas características, esta tecnología es utilizada principalmente en computadoras de escritorio y laptops [33].

2. **eMMC (Embedded MultiMedia Card)**: Esta también consta de un controlador y memoria empaquetada en un mismo paquete, pero brinda un alternativa más barata a los SSDs con un consumo de energía muy bajo. Estas características hacen que sea el estándar de facto como medio de almacenamiento primario en dispositivos móviles hoy en día [23].

3. **UFS (Universal Flash System)**: Es una especificación de almacenamiento flash muy reciente diseñada para su uso en dispositivos móviles. Brinda varias mejoras con respecto a eMMC y se perfila como su sucesor [22]. Prevén que será la tecnología de almacenamiento móvil dominante para fines de 2016 [44].

4. **SD (Secure Digital)**: Al igual que las anteriores, empaqueta un controlador y memoria flash aunque en un formato muy pequeño, lo cual las haces muy portables. A diferencia de las anteriores, la SD Cards están diseñadas para ser fácilmente removibles. Actualmente son el estándar de facto para tarjetas de expansión de dispositivos móviles [39].

1.2.4. Redes inalámbricas

Las redes inalámbricas conforman un aspecto esencial de los dispositivos móviles ya que le permiten al dispositivo tener un alto intercambio de datos con el exterior, particularmente, brindándole acceso a Internet. A continuación, veremos diversos tipos de redes con las que suelen contar la mayoría de los dispositivos móviles.

1.2.4.1. Redes celulares

Estas son las redes con las que solemos asociar inmediatamente a los dispositivos móviles, ya que ofrecen el mayor rango en términos de distancia proporcionándole así a este tipo de dispositivos su característica distintiva: movilidad.

Existen dos tipos de redes celulares dominantes hoy en día: GSM y CDMA [11]. Ambas han tenido su evolución tecnológica a lo largo de los años, brindando mayores capacidades hasta su más reciente evolución la cual se conoce como 4G (cuarta generación) [34]. Lo importante a tener en cuenta es que si bien la infraestructura requerida por los proveedores de servicios de ambas tecnologías es similar, la forma en que funcionan desde el punto de vista de la interpretación de la señal electromagnética es muy diferente. Esto implica que el hardware requerido para comunicarse con un tipo de red o la otra sea distinto. Por lo tanto, un modelo de celular sólo funciona para una de esas tecnologías generalmente. Aquellos que funcionan para ambas tecnologías contienen dos radios y se suelen llamar teléfonos globales o mundiales.

Los dispositivos móviles suelen ser obtenidos directamente del proveedor local de servicio de red celular al cual el usuario desea suscribirse. Sin embargo, los dispositivos móviles también pueden obtenerse en tiendas o comercios y suscribirse a un proveedor de servicio posteriormente. Aquellos dispositivos móviles que no se encuentran bloqueados para ser utilizados con un proveedor de servicio específico se los conoce como ünlocked". Los mismos pueden ser utilizados con diversos proveedores simplemente cambiando de SIM card.

Desde el punto de vista de la infraestructura, las redes celulares cubren grandes áreas geográficas dividiendo las mismas en çélulas". La cobertura de cada una de estas células la da una estación base que emite ondas de radios y se comunica con los dispositivos móviles en su rango.

Este aspecto es fundamental a cómo funcionan. Es necesario monitorear las

conexiones activas de un dispositivo móvil ya que a medida que el mismo se mueve de una célula a otra, dichas conexiones deben ser mantenidas. Los datos de por dónde transitó un dispositivo móvil son mantenidos por el proveedor de servicio. Además, para cobrar y brindar servicios acordes a lo contratado, los proveedores deben mantener datos acerca de su contrato y el uso de la red que éste realiza.

Los proveedores de servicio mantienen una base de datos llamada Home Location Register (HLR) que actúa como repositorio central de toda la información de sus suscriptores e información de sus servicios. También mantienen una base de datos aparte llamada Visitor Location Register (VLC) que es utilizada para mantener la información de uso de la red de dispositivos móviles que no pertenecen a la red del proveedor pero pagan para hacer uso de la misma (roaming). El HLR mantiene datos valiosos como datos de la cuenta del suscriptor, metadatos de llamadas, etc que suelen ser valiosos en investigaciones criminales.

1.2.4.2. WiFi

Las redes inalámbricas locales han probado ser muy populares. Hoy en día en áreas urbanas nos encontramos inmersos en un mar de puntos de acceso inalámbricos. Los mismos se los puede encontrar desde el hogar hasta la oficina y prácticamente en cualquier espacio público.

Los dispositivos móviles dan preferencia al uso de WiFi sobre red celular siempre que las primeras se encuentren disponibles. Esto se debe a que generalmente ofrecen una señal más estable y tienden a brindar un mayor ancho de banda, pero en particular, también por un tema económico dado que las conexiones de datos celulares suelen tener límites de tráfico de datos.

Cada dispositivo móvil cuenta con una dirección MAC que el fabricante le asigna de forma de identificar dicha interfaz de forma única. Esto brinda oportunidades interesantes en el cual se puede saber potencialmente con qué otros dispositivos de red inalámbrica estuvo interactuando un dispositivo móvil.

Lo anterior es aplicado por sistemas de posicionamiento que tienen en cuenta los puntos de acceso WiFi. Dada la gran ubicuidad de esta tecnología hoy en día y el hecho de cada punto de acceso hace un broadcast de su SSID y dirección MAC (qué es única), varias compañías tecnológicas vieron la posibilidad de recolectar esta información de sus usuarios y almacenarla de

forma masiva [55]. De esta forma, cuentan con una asociación entre posición geográfica y dirección MAC del correspondiente punto de acceso. Empresas que brindan servicios de geolocalización como Apple, Google y Microsoft son capaces de ubicar dónde se encuentra un dispositivo únicamente conociendo los puntos de acceso que el dispositivo percibe en cierto momento (esto es, los beacons que el mismo está recibiendo). Esta funcionalidad beneficia al usuario del servicio ya que le brinda un gran nivel de precisión al servicio de geolocalización, así como también la posibilidad de utilizar el servicio en lugares donde no es posible utilizar el GPS del dispositivo [52].

1.2.4.3. Bluetooth

Por otro lado, Bluetooth es una tecnología inalámbrica muy utilizada para intercambiar datos sobre distancias cortas, típicamente hasta varios metros. Tiene varias aplicaciones muy interesantes como es su uso en accesorios inalámbricos, intercambio de archivos entre dispositivos móviles cercanos, etc.

Más recientemente, gracias al desarrollo de Bluetooth LE (Low Energy) se le está dando un uso particular en un dispositivo el cual Apple denomina como 'iBeacon'. El llamado iBeacon es tan sólo un dispositivo bluetooth que emite un beacon con cierto identificador único (UUID). Esto posibilita una serie de usos que toman en cuenta el contexto local en que se encuentra un dispositivo. Por ejemplo, en una tienda se pueden colocar iBeacons, de forma que usuarios que tengan la aplicación de la tienda, le aparezcan notificaciones a medida que se mueve por diversas secciones de la tienda. De esta forma, la aplicación le puede proporcionar información contextual, por ejemplo, para ayudar al cliente en la navegación del local físico o brindan su atención hacia productos en promoción [50].

Por lo tanto, vemos que mediante Bluetooth también es potencialmente posible recolectar información acerca de donde estuvo un dispositivo mediante el uso de iBeacons, o con qué otros dispositivos Bluetooth interactuó.

1.3. Aspectos de software

Ya vimos la gran diferencia que existe entre el software presente en smartphones con respecto al de feature phones. En esta sección nuestra intención es ver en más detalle los aspectos que caracterizan al software de los smartphones.

1.3.1. Sistema operativo

Hoy en día, todos los sistemas operativos que encontramos en smartphones tienen capacidades comparables a los tradicionales sistema operativos de PC. Sin embargo, además de presentar grandes cambios en cuanto al diseño de interacción con el usuario, estos sistemas presentan varios aspectos técnicos importantes que los diferencian de los sistemas anteriores. Veamos algunos de estos aspectos que son de interés:

1. Uso eficiente de los recursos limitados con los que cuentan.
   - **Poder de procesamiento:** Si bien los dispositivos móviles cada vez cuentan con un mayor poder de procesamiento, la brecha entre el procesamiento con el que cuentan los PCs hoy en día aún es considerable.
   - **Almacenamiento:** De forma similar, la capacidad de almacenamiento ha aumentado considerablemente, pero es significativamente menor con la que podemos contar en las PCs, por lo cual tanto aplicaciones como sistema operativo debe hacer un uso eficiente del mismo.
   - **Batería:** El ritmo con que avanza la tecnología informática no ha tenido su contrapartida en la tecnología de las baterías. La duración de las mismas no ha aumentado considerablemente. Por lo tanto, el mismo pasa a ser un recurso de mayor relevancia. Para aumentar la duración de la batería se debe intentar ser lo más eficiente posible con todos los recursos del dispositivo en general. Desde el punto de vista del software, esto quiere decir tener mecanismos dentro del sistema operativo como pueden ser disminuir el brillo de la pantalla en ciertas situaciones, matar procesos que hagan uso extenso de los recursos, etc. También implica que los desarrolladores de software deben tener especial cuidado al implementar funcionalidades en sus aplicaciones.
2. Los sistemas operativos móviles introducen varios conceptos de seguridad importantes a sus modelo de seguridad con respecto a los disponibles por defecto en sistemas operativos de PC. Veamos dos muy importantes que se encuentran en los dos sistemas más populares (Android e iOS):
   - **Sandboxing de aplicaciones:** El sistema operativo encapsula a cada aplicación imponiendo una capa a través de la cual la aplicación debe interactuar con el sistema operativo pasando controles

antes de acceder a recursos fuera de la misma.

- **Control de acceso a recursos:** Establecen un fuerte control de acceso a recursos basado en permisos. Mediante el mismo buscan comunicar de forma clara al usuario los recursos a los cuales tiene acceso cada aplicación.

3. El ciclo de liberación de versiones de los sistemas operativos móviles es considerablemente más rápido que al que estábamos acostumbrados para sistemas operativos de PC. iOS tiene un promedio de una nueva versión por año [48] mientras que para Android el promedio es menor a un año [46]. En los sistemas operativos de escritorio el promedio de tiempo entre nuevas versiones rondaba los cuatro años [49].

4. Por último, como hemos visto, existe una mayor diversidad de sistemas operativos y cambios en las cuotas de mercado de los mismos que en la industria de las PC. Hasta hace unos años, se dió una competencia muy intensa entre varios contendientes con el fin de capturar el mercado. En ese período surgieron muchos sistemas operativos móviles. Hoy en día tenemos dos claros vencedores: Android e iOS [18]. De todas formas, continúan surgiendo sistemas operativos móviles como es el caso de Windows Phone (de Microsoft), Tizen (de Samsung), Firefox OS (de Mozilla) y más. Por otro lado, otros que tuvieron gran participación del mercado en el pasado reciente como Blackberry OS, Symbian OS, etc han ido desapareciendo lentamente. Estos cambios en un lapso tan corto le da un aspecto de gran dinamismo a la industria de los dispositivos móviles. Para herramientas que dependen de estas plataformas, esto significa que las mismas en general deben estar en constante actualización.

1.3.2. Aplicaciones

Empecemos por recordar que la distribución de aplicaciones en PC tradicionalmente se ha dado de forma descentralizada y los controles que se realizan antes de instalar una aplicación de escritorio suelen ser mínimos por parte del sistema operativo. Veamos cómo el ecosistema de aplicaciones que existe en el mundo móvil es notoriamente distinto.

En el modelo utilizado usualmente por las plataformas móviles se cuenta con una tienda virtual que centraliza el descubrimiento y distribución de aplicaciones. Esto además viene acompañado de un mecanismo que facilita al

usuario la instalación y actualización de las mismas. Esta capacidad centralizada le brinda varias ventajas importantes a la organización responsable de la plataforma.

Por un lado, tiene la capacidad de controlar varios aspectos del proceso, entre ellos, cómo los usuarios descubren las aplicaciones, qué tipo de contenido consideran apropiado distribuir, para cuáles regiones geográficas desean que esté disponible cierto contenido, y muchos más. Además, pueden imponer ciertos requerimientos de calidad sobre las aplicaciones.

Por otro lado, tiene la capacidad de implantar mecanismos que le permitan mitigar la distribución de malware y reducir de esta forma significativamente el impacto en los usuarios de la plataforma.

Capítulo 2

Tipos de datos

A pesar de las dificultades que suelen presentar los dispositivos móviles a la hora de extraer datos, estos son una excelente fuente de datos ya que cuentan con una gran cantidad de información, mucha de la cual no solemos encontrar en otros tipos de dispositivos como PCs.

Su naturaleza personal hace que podamos establecer una relación de uno a uno entre el dispositivo y su propietario. Esto facilita la atribución de acciones a un individuo. En investigaciones forenses dicha relación resulta clave al permitir establecer la última milla de evidencia que la relaciona a un individuo [11].

Tras haber visto las características que consideramos más relevantes de los dispositivos móviles, intentaremos identificar los diversos tipos de datos que podemos encontrar en los mismos y algunos ejemplos de los usos que se les puede dar en investigaciones forenses.

Hoy en día ya no contamos con un conjunto de tipos de datos finitos y previsibles como era usual al examinar feature phones los cuales contaban con unos pocos tipos de datos como contactos, mensajes, fechas de calendario, etc. Mediante el modelo adoptado por las plataformas móviles, los dispositivos cuentan con acceso de forma muy sencilla a una inmensa cantidad de aplicaciones para una gran diversidad de usos. Esto hace que podamos llegar a encontrarnos con tipos de datos de toda índole.

De esta forma, tendremos que estar preparados para enfrentarnos a una gran diversidad de tipos de datos. Empecemos entonces por ver varios de los tipos

de datos que solemos encontrar en las fuentes de datos típicas que brindan estos dispositivos.

2.1. Telefonía

Como ya vimos, los dispositivos móviles cuentan con la capacidad de conectarse a las redes celulares. Para poder hacer uso de los servicios brindados por la red los dispositivos deben autenticarse. Para ello cuentan con identificadores que son asignados por el proveedor de servicio. En redes GSM, este corresponde al IMEI (International Mobile Equipment Identifier), mientras que en redes CDMA, es el MEID (Mobile Equipment Identifier) [11].

Ya vimos también que los dispositivos GSM cuentan con una tarjeta extraíble comúnmente llamada SIM Card. La misma tiene la capacidad de almacenar datos de usuario como contactos, registro de llamadas, SMS y más. Por otro lado, también contiene datos del servicio, que además del identificador del dispositivo en la red, contiene otros datos como una lista de LAIs (Location Area Identifier). Esta lista contiene los identificadores de las últimas áreas desde las cuales se conectó a la red el dispositivo. De esta manera, cuando el dispositivo cambia de área geográfica, un nuevo LAI es agregado a la lista.

Veamos algunos usos que se le podría dar esta información en un análisis forense [21]:

- Utilizando los identificadores de telefonía (MEID ó IMEI según la red), los proveedores de servicio de telefonía almacenan la información del propietario del dispositivo. Mediante una orden judicial, se podría obtener esta información para establecer la relación entre un dispositivo encontrado y la persona por la que fue utilizado.
- Como vimos, la lista de LAI almacenada en la SIM nos permite conocer las zonas geográficas que visitó un dispositivo. Con esta información podemos saber por donde estuvo un individuo y corroborar información que éste afirme.
- Otro dato interesante que podemos encontrar es el LND (Last Number Dialed). Dependiendo del dispositivo, el mismo es almacenado en la SIM card o en el almacenamiento interno del dispositivo. Como su nombre lo indica, el mismo puede ser usado para identificar al último número marcado desde el dispositivo, lo cual nos permite encontrar relaciones entre sucesos ocurridos.

2.2. PIM

Los tipos de datos de lo que se conoce como PIM (Personal Information Management) abarcan contactos, registro de llamadas, calendario y recordatorios. Estas funcionalidades surgieron con los feature phones, y fueron mejorando en su evolución. Algunas de las diferencias que podemos observar en estos tipos de datos entre feature phones y smartphones son:

- Usualmente los feature phones almacenan contactos con unos pocos campos como nombre, telefono y dirección, mientras que los smartphones almacenan una mayor cantidad, entre los cuales se suele encontrar email, foto, teléfonos extra, tono de llamada, etc.
- En los feature phones el calendario es almacenado de forma local, mientras que en los smartphones existe la posibilidad de sincronizar el calendario con distintas cuentas como Gmail y Outlook, entre otros.
- En los smartphones, cada uno de estos tipos de datos son manejados por una aplicación distinta, las cuales almacenan los datos dentro de su espacio reservado.

Utilizando los contactos de la agenda y los contactos de las redes sociales almacenados en el dispositivo es posible recrear la red social del propietario del dispositivo [26] y observar su vínculo con otros individuos. El registro de llamadas suele complementar este análisis, ya que contiene todas las llamadas realizadas, recibidas y perdidas en el dispositivo. Es posible que un número no esté guardado en los contactos, pero sí hayan llamadas realizadas hacia o desde éste, creando un vínculo con el mismo. A partir de las entradas del calendario, es posible determinar las acciones realizadas por el propietario del dispositivo y recrear una línea de tiempo de eventos.

2.3. Mensajería

Los SMS son una via rápida para comunicarse con otros dispositivos. Es una funcionalidad soportada tanto por los feature phones como por los smartphones. Dentro de la información contenida en cada SMS se encuentran el número de remitente, el número de destinatario, la fecha de envío, el tiempo de validez del mensaje (utilizado por la red de telefonía en el procesamiento del mensaje), el número del centro de control originario del mensaje (SMSC), y el propio texto del mensaje [51].

Los MMS (Multimedia Messaging System) permiten a los dispositivos móviles el envio y recepción de contenidos multimedia (imagen, audio ó video).

Además de vincular al propietario del dispositivo con el destinatario de un SMS o MMS, es posible obtener el contenido del mensaje (texto en el caso de SMS o MMS, y fotos, videos o sonidos en el caso de MMS), lo cual no es posible con las llamadas ya que su contenido no queda almacenado en el dispositivo. Estos datos están almacenados dentro del espacio reservado para la aplicación de mensajes del dispositivo, pudiéndose almacenar los archivos multimedia en otros directorios a elección del usuario. En el caso de los dispositivos GSM, los SMS también pueden almacenarse dentro de la SIM.

A diferencia de las llamadas, los datos de marcas de tiempos de estos mensajes suelen ser establecidos por el proveedor de servicios y no por el dispositivo, lo cual hace que estos sean más confiables. Por otro lado, este tipo de mensajes tiene la desventaja de no contar con datos sobre cuándo fue leído por primera vez.

2.4. Redes sociales

Dado el gran uso cotidiano de las aplicaciones de redes sociales, estas son una gran fuente de información en las cuales podemos encontrar una variedad de tipos de datos que pueden resultar de interés. La gran mayoría de las aplicaciones de redes sociales mantienen información sobre el usuario, sus conocidos, gustos, relaciones, eventos a los que asistió, publicaciones realizadas, mensajes enviados y recibidos, etc.

Como mencionamos anteriormente, la información de contactos de redes sociales puede ser utilizada para trazar vínculos del propietario del dispositivo con otros individuos. Además, dada la gran cantidad de actividades del usuario que es registrada es posible recrear una línea de tiempo con las actividades del individuo [26].

2.5. Geolocalización

Uno de los componentes más revolucionarios de los dispositivos de hoy en día es el GPS. Poder saber las coordenadas geográficas del dispositivo es un dato de suma importancia. Aplicaciones de geoposicionamiento como Google Maps mantiene posiciones del dispositivo en sus bases de datos internas. Entre esta

información se encuentran las coordenadas de los destinos buscadas por el usuario, junto con sus respectivas marcas de tiempo [28].

Utilizando este tipo de información, se pueden recrear las rutas realizadas por el usuario del dispositivo.

2.6. Metadatos

Además de las coordenadas almacenadas por aplicaciones de geolocalización como acabamos de ver, la cámara del dispositivo también suele incluir en los metadatos del archivo producido las coordenadas que indicaba el GPS en el momento que se tomó la foto o video junto a su correspondiente timestamp [54].

Además de los distintos tipos de archivos encontrados en los dispositivos móviles, existe información asociada a estos la cual debe ser analizada.

Así como podemos encontrar metadatos sobre dónde, cuándo y cómo fue obtenido un archivo multimedia, hay una gran variedad de otros metadatos que suelen ser producidos por los dispositivos móviles.

Vemos otro caso que es bastante usual. De igual forma en las PC, los navegadores web de los dispositivos móviles almacenan metadatos sobre las navegaciones realizadas por el usuario. Esta información es guardada para mejorar la experiencia del mismo, pero puede ser utilizada por nosotros para investigar su actividad en la web. Entre los datos almacenados se encuentran páginas visitadas, cookies, bookmarks, información de pagos por internet, etc.

2.7. Sistema operativo

Si consideramos la información mantenida por el sistema operativo del dispositivo, podemos identificar una diversidad de tipos de datos interesantes.

Dada la gran cantidad de dispositivos que son usados hoy en día y la información sensible que manejan, los mismos se han convertido en un objetivo muy tentador de terceros malintencionados. De esta forma, uno de los usos que le podemos dar al hecho de conocer las aplicaciones instaladas en un dispositivo es identificar si el mismo posee una aplicación maliciosa comparando

las aplicaciones instaladas con una lista de aplicaciones maliciosas conocidas [12].

Un tipo de malware particularmente peligroso en dispositivos móviles son los troyanos. Un uso muy común que suelen explotar es el de spammer, enviando correos por ejemplo. En el caso de dispositivos móviles, donde los recursos de telefonía son costosos tanto del punto de vista económico como en relación a la duración de la batería, el constante envío de datos que puede provocar este tipo de malware tiene un impacto significativo. Es conocido el hecho de aplicaciones malintencionadas que contienen comportamiento oculto (troyano) que se dedican a enviar SMS premium para incurrir en ganancia a cierta parte [40].

Por otra parte, los logs del sistema operativo son una fuente de datos interesante. Los sistemas operativos como Android e iOS están basados en los sistemas operativos Linux y BSD respectivamente. Por lo tanto es esperable que existan muchos archivos de logs propios de estos sistemas. Mucha información relacionada a las aplicaciones instaladas y los servicios actuales en ejecución en el dispositivo puede ser obtenida utilizando los archivos de logs del sistema operativo.

2.8. Servicio celular

Los proveedores de servicio mantienen información acerca de llamadas como fecha, duración, clasificación entrantes/salientes, recibidas/perdidas/correo de voz, y las antenas utilizadas durante la misma. Este tipo de datos si bien no corresponde a la extracción de datos del dispositivo en sí, puede ser de suma importancia. Utilizando estos datos es posible localizar al dispositivo a lo largo de toda la llamada, ya que el mismo puede haber estado moviéndose.

Debido al funcionamiento del protocolo de IP móvil, es posible localizar al dispositivo mediante las conexiones a Internet, tanto dentro como fuera de la red donde se estableció la conexión.

En los dispositivos GSM, es posible solicitar el PUK al proveedor de servicios, de modo de desbloquear la tarjeta SIM.

Para obtener estos datos es necesario contar con una orden judicial mediante la cual se solicita la cooperación del proveedor de servicios. Los datos del proveedor pueden ser utilizados para corroborar los obtenidos del dispositivo.

Capítulo 3

Tipos de métodos de extracción

A la hora de extraer los datos de un dispositivo móvil es necesario tener en cuenta las características del método de extracción a utilizar. Estas características hacen referencia tanto a las capacidades de los métodos a la hora de obtener los datos como a las consecuencias que los mismos tengan sobre el dispositivo [10].

En la siguiente figura se puede ver los distintos tipos de métodos de extracción:

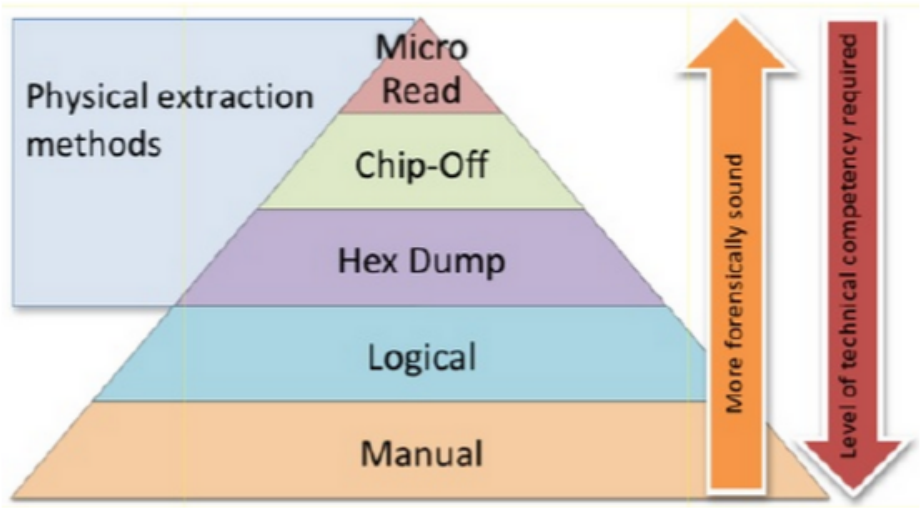

Figura 3.1: Tipos de métodos de extracción [9]

A medida que nos movemos hacia la cima de la pirámide, los métodos:

- Resultan más limpios desde el punto de vista forense.
- Requieren de utilizar herramientas más caras.
- Requieren mayor conocimiento técnico.
- Exigen más tiempo para analizar los datos extraídos.
- Requieren más entrenamiento.
- Son más invasivos.

Comenzando por la base de la pirámide, los métodos *manuales* requieren interactuar con el dispositivo utilizando el teclado o pantalla táctil del mismo para obtener los datos. Esto hace que este tipo de métodos sea soportado por una gran cantidad de dispositivos. Dado que la interacción con el dispositivo se da a través del sistema operativo, únicamente se pueden obtener datos que estén disponibles a través del mismo [11].

En el segundo nivel se encuentran los métodos de extracción *lógicos*, los cuales consisten en conectar el dispositivo a una PC y utilizar algún protocolo de intercambio de datos para obtener los datos del dispositivo. Al igual que con los métodos manuales, los únicos datos que se pueden obtener son los accesibles a través del sistema operativo. Son métodos rápidos, automatizables y la interpretación de los datos obtenidos cuenta con las abstracciones del sistema operativo.

En el tercer nivel tenemos los métodos de extracción *físicos*. Estos consisten en copiar de forma parcial o total los datos del dispositivo en su forma bruta (raw data). A diferencia de los métodos manuales y lógicos, los métodos físicos no interactúan con el sistema operativo del dispositivo. Debido a esto, se puede obtener la totalidad de los datos del dispositivo (incluyendo datos borrados), pero hay que tener en cuenta que estos datos deberán ser interpretados.

En el cuarto nivel se encuentran los métodos conocidos como *chip-off*. Como lo dice su nombre, estos métodos consisten en remover físicamente el chip de memoria del dispositivo, y luego (utilizando equipos especializados) realizar la lectura del contenido del chip de forma directa. Al igual que los métodos del nivel anterior, estos métodos permiten obtener la totalidad de los datos del dispositivo. Los aspectos negativos más importantes son el hecho de que el dispositivo queda inoperante y los datos obtenidos deben ser interpretados.

En el último nivel se encuentran los métodos *micro read*. En estos métodos se utilizan microscopios poderosos para ver el estado de la memoria del dispositivo, siendo los mismos los más difíciles y caros de llevar a cabo.

Capítulo 4

Representación de los datos extraídos

Tras realizar la extracción de datos, hay dos aspectos muy importantes que determinarán cómo trabajaremos posteriormente con los datos obtenidos. El primero se trata de qué formato utilizaremos para almacenar los datos extraídos, mientras que el segundo es el lenguaje que utilizaremos para intercambiar tanto los datos obtenidos como la metadata que fuimos asociando a los mismos en el proceso de extracción e incluso en etapas de examinación posteriores.

El formato en que almacenemos los datos extraídos determinará que otros datos son incluidos sobre los mismos (como puede ser hashes del contenido, etc) y qué facilidades nos brinda (como puede ser compresión del contenido y la posibilidad de separarlo en varios archivos).

Los lenguajes de intercambio nos ayudan a expresar los datos relevantes a la investigación de forma que esta pueda ser compartida. Esto puede resultar de inmensa importancia cuando se requiere trabajar con varios equipos dado que facilita la colaboración. Además, nos permite automatizar procesos, y en la medida que el lenguaje se vuelve estándar la interoperabilidad entre herramientas se hace mucho más fácil.

En ambos casos las decisiones que tomemos limitarán las herramientas con las que podremos trabajar con facilidad. Uno de los inconvenientes presentes hoy en día en la comunidad forense es el hecho de que cada herramienta comercial utiliza sus propios formatos propietarios y esto hace que la inter-

operabilidad entre herramientas sea difícil. Por esto, resultan de gran importancia las iniciativas que están surgiendo de formatos abiertos, y los esfuerzos de estandarización [35].

En [1] se realiza un estudio comparativo de las diversas alternativas que hay en ambos rubros en 2010. A continuación, en la sección de formatos de almacenamiento mencionaremos los lenguajes relevantes que existen hoy en día y son de interés en cierto grado para el trabajo, y en las siguientes dos secciones analizaremos en detalle los dos lenguajes que consideramos que tienen la mayor relevancia: CybOX y DFXML.

4.1. Formatos de almacenamiento

Para expresar los datos obtenidos como resultado de una extracción física, se suele utilizar lo que se conoce como una disk image.

Una disk image refiere a un archivo (o más) que contiene una copia bit a bit del contenido y estructuras de la totalidad (o una porción) de un dispositivo de almacenamiento (como puede ser un disco duro o una memoria flash). De esta forma, se obtiene una réplica de los datos del dispositivo de almacenamiento de forma independiente al filesystem [47].

Este procedimiento es una práctica estándar que se suele realizar de forma previa a cualquier análisis forense, siempre que sea posible. El mismo resulta importante ya que permite [16]:

- Asegurar que la información no es modificada de forma involuntaria durante el análisis.
- Reproducir los resultados obtenidos en un análisis.
- Capturar información que no es visible al sistema operativo.
- Evitar la manipulación del dispositivo de almacenamiento físico, facilitando la distribución de su contenido de forma digital.

Con respecto a los formatos utilizados para representar a las disk images, existe una gran variedad, tanto estándar y abiertos como propietarios. Veamos a continuación varios de los más destacados que son utilizados con fines forenses. Cabe mencionar que existen varios formatos de disk images populares, como por ejemplo ISO, DMG de Apple y VMDK de VMWare, que omitiremos verlos dado que están más orientados a otros fines.

4.1.1. Raw Format (.dd / .img)

Se trata del formato más básico utilizado. Consiste en únicamente una copia bit a bit del contenido del dispositivo, sin ningún tipo de metadata agregada. De todas formas, generalmente en casos que se utiliza este formato también es usual almacenar metadatos en otros archivos separados [17].

La herramienta más antigua pero muy conocida que utiliza este formato es el comando Unix dd [15]. Posteriormente surgieron dos forks de dd, dcfldd [14] y dc3dd [13], brindando mejoras del punto de vista forense. Por ejemplo, agregándole funcionalidades como hashing on-the-fly y la posibilidad de dividir la salida en varios archivos. En particular, dc3dd permite contar con el hash de cada bloque, lo que posibilita comparar dos disk images tomadas de un mismo medio y observar en qué regiones se produjeron cambios.

4.1.2. Expert Witness Format (EWF)

Se trata de un formato propietario utilizado por la suite de productos de EnCase. El mismo cuenta con un amplio reconocimiento a nivel mundial en contextos judiciales, haciéndolo el actual estándar de facto en el área forense [25].

Si bien el formato no es abierto, LibEWF [24] es una implementación open source desarrollada por terceros que permite acceder al formato EWF y que es utilizada por varias herramientas.

4.1.3. Advanced Forensic Format (AFF)

Este formato surge con la iniciativa de crear un formato abierto y extensible para el uso de disk images teniendo en cuenta buenas propiedades forenses [38].

La evolución de este formato, AFF4 [27], provee una gran capacidad de abstracción que permite no sólo trabajar con múltiples tipos de evidencia más allá de disk images sino que permite trabajar de forma distribuida. Esto resulta importante para conjuntos de datos muy grandes e investigaciones que involucran la coordinación de varios equipos geográficamente distribuidos.

Tras haber visto estos formatos, es preciso mencionar que existen también una gran variedad de otros formatos. La mayor parte de las suites comerciales forenses tienen su propio formato propietario, de forma similar a EnCase con EWF. Mientras que por el lado de la investigación, han surgido algunas propuestas de formatos como Digital Evidence Bags (DEB) [37] con ideas muy interesantes pero que no han visto un posterior desarrollo y adopción.

4.2. Lenguajes de intercambio

En lenguajes de intercambio de datos de evidencia forense han surgido varios interesantes. En particular, nosotros vamos a considerar dos (CybOX y DFXML) por las características que veremos que presentan en las siguientes secciones. En particular, ambos lenguajes son abiertos y tienen un desarrollo activo del mismo y sus herramientas.

De todas formas, cabe mencionar dos lenguajes, **DEX** (Digital Evidence Exchange) [8] y **XIRAF** [45], que plantean ideas interesantes pero que no consideraremos ya que carecen alguna de las dos características antes mencionadas.

El primero permite que a partir de una raw image y una descripción, la evidencia pueda ser obtenida por múltiples herramientas de forma automatizada. Esto posibilita la capacidad de poder reproducir el proceso de obtener la evidencia y la verificación de herramientas forenses. También permite encontrar diferencias entre datos obtenidos por herramientas distintas sobre una misma fuente.

El segundo trata de un framework, desarrollado por el Instituto Forense de Holanda, que busca establecer una clara separación entre la extracción y el análisis de los datos, tener un único formato para entrada de las herramientas forenses y por último contar con un lenguaje para almacenar las salidas de los análisis y realizar consultas sobre éstas. Si bien se encuentra disponible un paper sobre la investigación, el framework y su especificación no se encuentran disponibles públicamente.

4.2.1. CybOX

CybOX [31] es un lenguaje estructurado para representar lo que se denominan *cyber observables*. Estos son eventos o propiedades con estado que pueden

ser observados en el dominio cibernético. El lenguaje no busca satisfacer un caso de uso en particular, sino que intenta ser lo suficientemente flexible para prestar una solución común a un amplio espectro de casos de uso de ciberseguridad que requieran manejar cyber observables.

CybOX soporta un amplio espectro de dominios de ciberseguridad incluyendo caracterización de amenazas, caracterización de malware, manejo operacional de eventos, respuesta de incidentes, intercambio de indicadores de compromiso, digital forensics y más.

De esta forma, vemos que CybOX es utilizado para una gran diversidad de finalidades. En cada una de estas el mismo generalmente es utilizado de forma indirecta, esto es, a través de otro lenguaje que aprovecha su expresividad para describir cyber observables en su dominio.

4.2.1.1. ¿Qué intenta resolver?

Podemos considerar tres principios que caracterizan las intenciones del lenguaje [30]:

1. **No está dirigido a un caso de uso de ciberseguridad en particular**: La intención es que sea lo suficientemente flexible para ofrecer una solución común para todos los casos de uso de ciberseguridad que requieran la capacidad de trabajar con cyber observables. Por lo tanto, busca principalmente ser una infraestructura para representar esta información, sobre la cual puedan construirse otros estándares y soluciones en cada dominio de trabajo específico.
2. **Flexibilidad tanto para expresar instancias como potenciales patrones**: También busca ser lo suficientemente flexible para permitir describir tanto instancias de cyber observables de alto grado de fidelidad como patrones más abstractos de potenciales cyber observables.
3. **Visión de automatización integrada**: Al tener especificado un esquema común y estructurado para los cyber observables, es posible desarrollar la capacidad de automatizar el intercambio de información detallada de los mismos.

4.2.1.2. Componentes de CybOX

CybOX presenta un modelo de datos separado en cuatro componentes:

Core: Consiste en los esquemas CybOX Core y CybOX Common que definen las construcciones básicas del lenguaje (como *Observable* y *Property*) y tipos de datos comunes utilizados por objetos (como *HashType* y *TimeType*).

Objects: Consiste de todos los esquemas de los objetos predefinidos que vienen por defecto con CybOX. Entre estos podemos encontrar Device, File, URI, User Account, etc.

Vocabularies: Consiste en varias listas de valores definidos que suelen resultar útiles para el uso por parte de los objetos. Por ejemplo, *HashNameEnum* contiene una lista de algoritmos de hashing como MD5, SHA256, etc. Es posible también crear uno mismo sus propios vocabularies.

Extensions: Consiste de esquemas externos que utiliza CybOX para representar algún aspecto como puede ser localización geográfica a través de CIQ o describir plataformas a través de CPE.

4.2.1.3. ¿Por qué y para qué nos sirve?

El lenguaje es de nuestro interés por las siguientes características que presenta:

- Su capacidad de representar propiedades con estado, hace que sea posible representar datos obtenidos en una extracción y metadatos de forma estructurada.
- El dominio de información sobre la cual trabaja incluye al de digital forensics. Muchos de los objetos predefinidos son relevantes a nuestro dominio y esto hace que podamos reutilizarlos.
- La visión de automatización y las diversas herramientas que proporciona para facilitar su procesamiento es un aspecto clave. Consideramos importante que la información producida en el lenguaje pueda ser utilizada por parte de otras herramientas con facilidad.
- El hecho de que sea un lenguaje estándar y el mismo sea utilizado por diversas comunidades es también importante. Indica la madurez del mismo y ayuda al soporte del mismo por parte otras herramientas.

4.2.1.4. Representación de tipos de datos

Como vimos, un aspecto interesante de CybOX es que el mismo nos brinda un conjunto de objetos predefinidos relevantes para el dominio de digital fo-

rensics. En su última versión a la fecha (2.1), podemos observar objetos como File, Email Message, SMS Message, URI, etc que nos permiten representar varios tipos de datos de interés en el contexto del trabajo.

Por otro lado, cabe destacar que también es posible extender el conjunto de objetos que nos permite representar CybOX (esto es, más allá de los predefinidos). En la siguiente sección, veremos cómo podemos utilizar un mecanismo que brinda CybOX para hacer esto.

4.2.1.5. Utilizando CustomObject

El objeto *CustomObject* nos permite especificar objetos que no tienen un schema definido. Esto hace que resulte fácil utilizarlos para representar nuevos objetos pero para esto ambos productor y consumidor deben ponerse de acuerdo en los campos declarados dado que no cuenta con un schema que defina el nombre y tipo de cada uno.

Veamos un ejemplo en el cual tenemos un *CustomObject* [29] que representa la contraseña de un documento de Microsoft Office:

```
<CustomObj:CustomObjectType xsi:type="CustomObj:CustomObjectType"
    custom_name="example:OfficePassword">
  <cyboxCommon:Custom_Properties>
    <cyboxCommon:Property name="password"
    description="MS Office encryption password">
      SuP3rS3cr3T!
    </cyboxCommon:Property>
  </cyboxCommon:Custom_Properties>
    <CustomObj:Description>
      This is a string used as a password to protect an Microsoft Office
          document.
    </CustomObj:Description>
</CustomObj:CustomObjectType>
```

4.2.1.6. Extensión del lenguaje

Desde el punto de vista técnico, CybOX es un lenguaje que está basado en XML y se encuentra definido utilizando XML Schema. Esto implica que debemos contar con buen conocimiento de estos lenguajes si deseamos extender CybOX.

Existen varios puntos de extensión del lenguaje CybOX. En particular, los más frecuentes suelen ser la definición de nuevos objetos y vocabularios. Nosotros vamos a concentrarnos en el más interesante para nosotros que es

extender el lenguaje mediante la definición de nuevos objetos, dado que nos permitan representar tipos de datos que no son considerados por el lenguaje actualmente.

Creando un nuevo objeto de CybOX
Para crear un nuevo objeto en CybOX es necesario extender el tipo abstracto *ObjectPropertiesType* que se encuentra definido en CybOX Common. Hay dos formas ligeramente distintas de hacer esto:

- Extender directamente de *ObjectPropertiesType*. Básicamente vamos a estar creando un objeto totalmente nuevo. Esta construcción nos brinda la capacidad de definir las propiedades del objeto.
- Extender indirectamente de *ObjectPropertiesType*. Esto es, no extender de *ObjectPropertiesType* sino de otro objeto CybOX. Este otro objeto a su vez extiende, ya sea directa o indirectamente, a *ObjectPropertiesType* dado que todos los objetos deben extender a éste. Esta forma nos permite reutilizar objetos CybOX, evitando duplicar formas de expresar lo mismo y teniendo un modelo de datos más coherente. Por ejemplo, podemos tener un objeto de tipo *FileObjectType*. Si queremos ser más específicos, podemos optar por un objeto *ImageFileObjectType* que extiende a *FileObjectType* y agrega algunas propiedades como formato de imagen, dimensiones de la imágen, etc.

A su vez, resulta importante tener en cuenta los recursos disponibles con los que contamos en CybOX Common y CybOX Vocabularies. En particular, todos los tipos de datos simples que solemos contar en la mayor parte de los lenguajes ya se encuentran definidos (excepto booleano) en CybOX Common. Dentro de CybOX Vocabularies, podemos encontrar varias listas de valores que nos pueden resultar útiles como nombres de algoritmos, estándares de encoding de caracteres, etc.

La documentación de CybOX [31] provee una guía que describe a grandes rasgos los pasos que debemos tomar cuando vamos a crear un nuevo objeto:

1. Decidir qué es lo que queremos representar en CybOX.
2. Determinar qué campos/atributos pueden ser utilizados para caracterizar el objeto CybOX.
3. Establecer una correspondencia entre los tipos de datos de dichos campos y los tipos de propiedades disponibles en CybOX.
4. Revisar objetos CybOX existentes para ver si las capacidades que de-

finimos en los pasos anteriores ya no se encuentran soportadas por un objeto existente. Además, identificar objetos CybOX que soporten un subconjunto de las capacidades necesarias, de forma de evaluar extenderlo y cubrir las restantes.

5. Definir un namespace para el objeto.
6. Crear el esquema para el objeto.
7. Agregar documentación al esquema.

Ejemplo Veamos cómo podemos extender el Device Object de CybOX para que pueda representar un dispositivo más concreto: un teléfono celular. Por lo tanto, además de incluir descripción, fabricante, modelo, número de serie, versión de firmware, etc que ya incluye un Device Object, debemos agregarle IMSI, IMEI y ICCID. Aquí ya hicimos los primeros cuatro pasos.

A continuación debemos escoger un namespace para nuestro nuevo TelephoneDevice Object (`http://fing.edu.uy/objects#TelephoneDeviceObject-2`) y crear un archivo *Telephone_Device.xsd* que contenga el siguiente esquema:

```
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:cyboxCommon="http://cybox.mitre.org/common-2"
  xmlns:DeviceObj="http://cybox.mitre.org/objects#DeviceObject-2"
  xmlns:TelephoneDeviceObj="http://fing.edu.uy/objects#TelephoneDeviceObject-2"
  targetNamespace="http://fing.edu.uy/objects#TelephoneDeviceObject-2"
  version="1.0">
  <xs:import namespace="http://cybox.mitre.org/common-2"
      schemaLocation="../cybox_common.xsd"/>
  <xs:import namespace="http://cybox.mitre.org/objects#DeviceObject-2"
      schemaLocation="Device_Object.xsd"/>
  <xs:element name="Telephone_Device" type="DeviceObj:DeviceObjectType"/>

  <xs:complexType name="TelephoneDeviceObjectType">
    <xs:complexContent>
      <xs:extension base="DeviceObj:DeviceObjectType">
        <xs:sequence>
          <xs:element name="IMSI"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
          <xs:element name="IMEI"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
          <xs:element name="ICCID"
              type="cyboxCommon:StringObjectPropertyType" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Si bien es bastante verbosa la representación, lo único que tuvimos que hacer

conceptualmente fue extender el objeto Device de CybOX y agregarle las tres propiedades que queríamos (IMSI, IMEI, ICCID) como strings.

Finalmente, para verificar que el esquema sea válido podemos verificarlo con la herramienta `xmllint` (disponibles en sistemas basados en Unix) de la siguiente forma:

```
xmllint --noout --dtdvalid http://www.w3.org/2001/XMLSchema.dtd
    Telephone_Device.xsd
```

Capítulo 5

Plataforma Android

En esta sección estudiaremos las características más importantes a tener en cuenta para extraer datos de un dispositivo Android. Veremos la arquitectura de la plataforma y el modelo de seguridad impuesto por la misma. Luego, observaremos los datos que podemos encontrar en estos dispositivos y las estructuras en que éstos son almacenados. Finalmente, terminaremos mostrando los mecanismos que podemos utilizar para extraerlos.

5.1. Arquitectura de la plataforma

La plataforma Android cuenta con una arquitectura dividida en cuatro capas [43], como se muestra en la figura 5.1.

La capa inferior consiste en un kernel Linux modificado, el cual provee una capa de abstracción para interactuar con el hardware. Contiene todos los drivers necesarios para interactuar con los distintos componentes del hardware (cámara, teclado, Wifi, audio, etc).

La segunda capa consta de dos secciones. Por un lado, un conjunto de bibliotecas específicas para cada componente de hardware, las cuales le permiten al dispositivo el manejo de un gran conjunto de datos. Por otro lado, el entorno de ejecución de Android, el cual provee un componente clave para la ejecución de las aplicaciones: *Dalvik Virtual Machine*. Cada aplicación tendrá su propia instancia de Dalvik VM, permitiendo que la aplicación ejecute dentro de un ambiente controlado e independiente. De esta forma, una aplicación no puede interferir indebidamente con otras aplicaciones, con el sistema operativo, ni

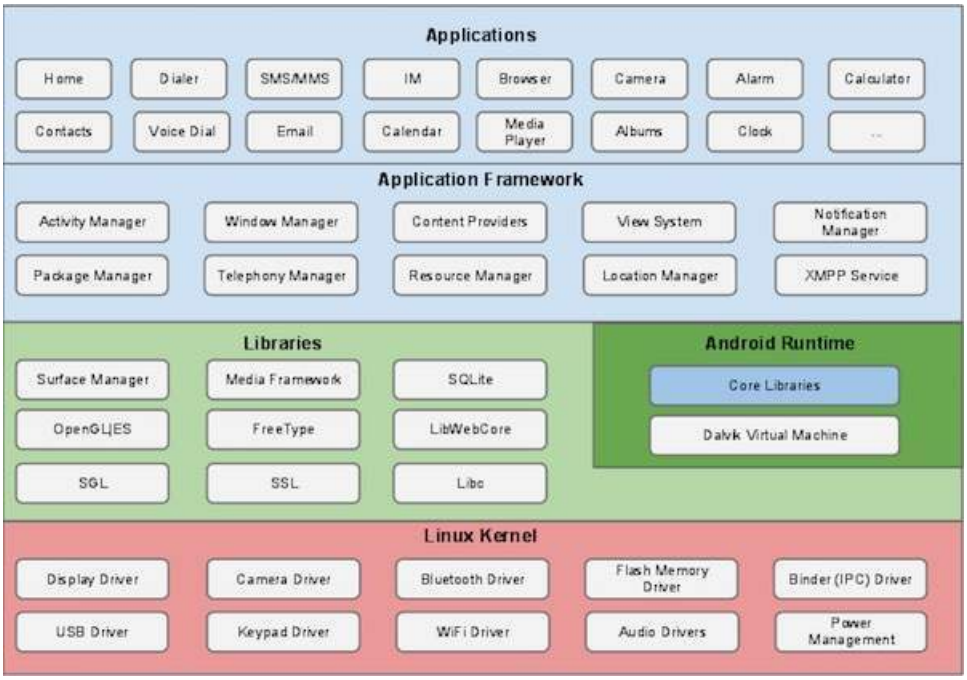

Figura 5.1: Arquitectura de la plataforma Android [5]

acceder directamente al hardware del dispositivo.

La tercera capa consta del *Application Framework*, encargado de brindar muchos de los servicios de alto nivel a las aplicaciones a través de clases java, para que los desarrolladores de aplicaciones puedan usarlos en las mismas. Por ejemplo, uno de estos servicios son los *Content Providers*, los cuales son utilizados por las aplicaciones para comunicar datos a otras aplicaciones.

Finalmente, en la cuarta capa se encuentran las aplicaciones instaladas en el dispositivo. Este conjunto de aplicaciones incluye tanto las pre-instaladas como las instaladas por el usuario.

5.2. Almacenamiento de datos

5.2.1. División habitual de las particiones

Generalmente la partición inicial / root, es una partición bastante chica. Hay un directorio especial */dev/* bajo el cual podemos encontrar archivos cuyo

único propósito es representar recursos que el sistema tiene disponibles (como suelen ser particiones lógicas). Para poder utilizar particiones lógicas que se encuentren bajo ese directorio, precisamos montarlas bajo en algún lugar bajo el directorio root. El lugar en el que se montan se le llama *mounting point*. Esto que hace que la partición sea accesible a través de ese "punto de montaje".

El esquema exacto de las particiones depende del modelo de dispositivo. Cada fabricante suele tener un esquema diferente. A continuación veamos a modo de ejemplo las particiones más importantes con las que cuenta uno de los modelos de dispositivos más populares actualmente, el Samsung Galaxy S5 [7]:

Device	Nombre	Filesystem	Montaje	Descripción
/dev/block/mmcblk0p15	boot	rootfs	/boot	Booteable (boot normal)
/dev/block/mmcblk0p24	cache	ext4	/cache	Archivos de cache
/dev/block/mmcblk0p12	efs	ext4	/efs	Datos del dispositivo
/dev/block/mmcblk0p16	recovery	ext4	n/a	Booteable (recovery mode)
/dev/block/mmcblk0p23	system	ext4	/system	Datos del sistema, aplicaciones pre-instaladas, etc (read-only)
/dev/block/mmcblk0p26	userdata	ext4	/data	Datos de usuario (apps)

Cuadro 5.1: Tabla Operations

Veamos el propósito de cada una de estas particiones [53] [20] que observamos en la tabla anterior.

Primero veamos aquellas particiones que se encuentran en memoria interna:

1. **Boot**: Utilizada para inicializar el dispositivo. Contiene el kernel y la ramdisk. Es la partición utilizada para iniciar el dispositivo, por lo que en caso de ser borrada, el mismo no podrá iniciar normalmente. Para instalar una nueva partición Boot, se puede instalar una ROM que la incluya.

2. **Cache**: Almacena los componentes que son usualmente utilizados con el objetivo de mejorar la experiencia del usuario. Si esta partición es borrada los datos del usuario no serán afectados, y con el uso del dispositivo estos datos borrados serán reconstruidos nuevamente.

3. **Efs**: Dedicada a almacenar información importante acerca del dispositivo, como por ejemplo información acerca de la señal de radio, direcciones MAC de Wifi y Bluetooth, IMEI, entre otros.

4. **Recovery**: Esta es una partición alternativa para bootear el dispositivo en Recovery Mode. Este modo es utilizado para realizar operaciones de mantenimiento como Factory Reset, borrar la partición de cache y actualizar manualmente el sistema operativo.

5. **System**: Contiene el sistema operativo, salvo el kernel y la ramdisk (sección de memoria RAM utilizada al inicializar el dispositivo). Incluye la interfaz gráfica y todas las aplicaciones del sistema que vienen preinstaladas. Si esta partición es borrada, se elimina completamente Android del dispositivo, pero este no queda inutilizable, ya que todavía se puede bootear en Recovery o Bootloader mode e instalar nuevamente una ROM.

6. **Data**: Contiene los datos del usuario, como por ejemplo contactos, mensajes, configuraciones y las aplicaciones instaladas por el mismo. Al hacer un Factory Reset, esta partición es borrada completamente, dejando al dispositivo en su estado original. Hay que tener en cuenta que esto no incluye al sistema operativo, ya que este se encuentra en la partición /system, por lo que las modificaciones realizadas al mismo no serán afectadas por un Factory Reset.

Ahora veremos las particiones que se encuentran en memoria externa:

1. **SdCard**: Se encuentra en la tarjeta SD. Los datos guardados en esta partición pueden ser accedidos por cualquier aplicación que cuente con permisos de acceder a la tarjeta SD, sin restringirse a los datos propios de cada aplicación. Borrar los datos de esta partición no afecta a los datos de las restantes particiones, solamente existe el riesgo de que hayan datos importantes usados por alguna aplicación, los cuales pueden ser borrados libremente. Existen dos tipos de tarjetas SD, internas y externas, y en caso de que existan ambas estas son conocidas como /sdcard y /sdcard/sd respectivamente.

2. **Sd-ext**: Esta partición no existe en los dispositivos Android con ROMs estándar. Son un agregado que tienen las custom ROM, las cuales son usadas como una extensión de la partición /data, de modo de incrementar el espacio donde se instalan las aplicaciones, pudiendo así los dispositivos con poco espacio de almacenamiento soportar la instalación de muchas aplicaciones. El borrar esta partición hace que se pierdan los datos de las distintas aplicaciones instaladas en la misma, al igual que si se borra la partición /data.

5.2.2. Montaje de particiones

Cuando el sistema inicia, las distintas particiones son montadas automáticamente en diversos puntos bajo el directorio root (/). De esta forma, los distintos puntos de montaje de las particiones (*/system*, */data*, */cache*, etc) son todos parte de la memoria interna del dispositivo, pero son separados en distintas particiones para que el sistema los vea como medios físicos diferentes. Lo mismo ocurre con cualquier medio físico externo, por ejemplo, la SD card es montada en */mnt/sdcard*. Cuando el dispositivo es apagado, las distintas particiones son desmontadas automáticamente por el sistema.

Como todo sistema Linux, Android mantiene un archivo llamado fstab, el cual es un mapeo de las distintas particiones lógicas con sus respectivos puntos de montaje y otros datos como las opciones que fueron utilizadas para realizar el montaje. Este archivo se usa en el proceso de booteo del dispositivo para montar las particiones, utilizando las distintas opciones indicadas en el mismo para cada una de las particiones. Entre las distintas opciones de montaje, las más importantes son read/write (rw) y read-only (ro). Estas configuraciones no son lo mismo que los permisos que tienen cada uno de los archivos y directorios para indicar quienes pueden leerlos y modificarlos. Las opciones con las cuales se montan las particiones tienen un nivel superior de jerarquía que los permisos de los archivos, ya que no va a ser posible modificar un archivo con permisos de escritura si la partición donde este se encuentra no fue montada con la opción read/write [4].

La partición */data* es montada con la opción read/write, ya que esto es necesario para poder instalar aplicaciones, cambiar configuraciones, crear archivos, etc.

La partición donde se encuentra el sistema operativo (*/system*) es montada automáticamente con la opción read-only al iniciar el dispositivo. Esto se

debe a que realizar modificaciones sobre esta partición es muy riesgoso. Aún teniendo root no es posible modificar una partición montada en modo sólo lectura.

5.2.3. Filesystems utilizados

Al igual que cualquier sistema Linux, existen diversos filesystems en uso en Android. Algunos de ellos son usados para bootear el dispositivo, correr el sistema o almacenar los datos del usuario.

Almacenamiento primario

Hasta Android 2.3 (Gingerbread), la versión Stock de Android utilizaba el filesystem YAFFS2. El mismo es un filesystem Open Source desarrollado especialmente para memorias flash NAND, el cual fue optimizado para dispositivos de bajas capacidades de memoria. Algunos de sus características interesantes son el contar con mecanismos de wear-leveling built-in para prolongar la vida útil de la memoria Flash y contar con robustez ante fallos de energía. Luego de Android 2.3, debido a que YAFFS2 no soporta multi-threading (entre otros factores [2]), la plataforma cambió el filesystem utilizado y pasó a usar EXT4.

Por otra parte, existen casos donde los fabricantes de dispositivos utilizan otros filesystems distintos a los anteriores. Por ejemplo, el Moto X utiliza F2FS para la partición de datos del usuario.

Almacenamiento extendido

Con el objetivo de brindar interoperabilidad entre sistemas operativos (Windows, Linux, Mac OS X), las SD Cards suelen utilizar filesystems FAT. Esta partición suele ser utilizada para permitirle a los usuarios conectar su dispositivo móvil a un PC y transferir archivos desde y hacia el mismo.

5.3. Información del sistema

Como ya vimos, Android fue creado sobre Linux, por lo que estas dos plataformas cuentan con características en común. Una característica muy útil para obtener datos del sistema, son los logs. Existen varios logs de los cuales

se pueden obtener datos del sistema, siendo log *dumpsys* el que brinda más información. Para acceder a este log primero es necesario que el dispositivo se encuentre en modo *debug*. Luego, debemos ejecutar el comando *adb shell dumpsys*.

Algunos de los datos que podemos encontrar son:

- **Cuentas sincronizadas**: la sección account contiene una lista con los emails utilizados para las cuentas sincronizadas, junto con la fecha y hora de la última sincronización de cada una de ellas.
- **Últimas coordenadas GPS conocidas**: la sección location contiene un conjunto de posiciones GPS.
- **Notificaciones del dispositivo**: la sección statusbar contiene un listado con las últimas notificaciones del dispositivo. En esta lista se muestra la aplicación que generó dicha notificación y alguna información extra sobre qué fue lo que se notificó. Por ejemplo, en caso de mensajes de WhatsApp se muestra quien envió el mensaje.
- **Información del sistema**: se puede obtener las versiones del bootloader y kernel, información sobre el proveedor de servicio de telefonía, número de build del sistema, entre otros.
- **Información de telefonía**: en la sección iphonesubinfo se puede encontrar el MEID y el ICCID del dispositivo.
- **Información sobre el uso del dispositivo**: en la sección usagestats se puede ver un listado con las últimas aplicaciones utilizadas y las diversas pantallas utilizadas dentro de las mismas.

5.4. Información de las aplicaciones

Las distintas formas de almacenamiento que pueden ser utilizadas por las aplicaciones son las siguientes [3]:

- **SharedPreferences**: Los SharedPreferences son archivos con formato XML, en los cuales el desarrollador de la aplicación puede almacenar un listado de parejas clave-valor de tipos de datos primitivos (booleans, floats, ints, longs y strings). Suelen ser utilizados para almacenar configuraciones de la aplicación, incluyendo los cambios de configuración realizados por el usuario. Estos datos son mantenidos a través de las sesiones del usuario, incluso cuando se cierra la aplicación.

- **SQLite databases**: Este tipo de bases de datos es el más utilizado por la comunidad Android, debido a que su código es Open Source, compacto y toda la base de datos se encuentra en un único archivo.
- **Otros tipos de archivos**: Es posible que las aplicaciones almacenen otros tipos de archivos (por ejemplo video, audio, texto, etc). Estos archivos pueden almacenarse o bien en memoria interna en un directorio que es privado para la aplicación, o bien en memoria externa (esto es, la SD-Card) en un directorio que es accesible por las demás aplicaciones.

La estructura de directorios con que cuenta una aplicación en su directorio privado en el almacenamiento de memoria interna es la siguiente:

- **databases**: directorio que contiene las bases de datos SQLite manejadas por la aplicación. Además de las bases de datos en sí, se pueden encontrar bases de datos temporales, las cuales son utilizadas por SQLite para realizar rollbacks.
- **cache**: directorio utilizado para almacenar archivos temporales. Estos datos pueden ser borrados automáticamente por la aplicación en caso de que no haya espacio suficiente en la memoria interna.
- **files**: directorio para almacenar archivos creados por la aplicación cuando la misma desea que los mismos no sean públicos para las demás aplicaciones.
- **shared_prefs**: directorio donde se almacenan los archivos SharedPreferences.

Cabe destacar que las aplicaciones también cuentan con la posibilidad de almacenar datos de forma remota a través de internet. Esto a menudo resulta importante cuando las aplicaciones desean evitar almacenar información sensible en el dispositivo.

5.5. Preparación del dispositivo para la extracción: Rooting

En el modelo de seguridad que impone Android, el dispositivo viene de fábrica en un estado en el cual el usuario del mismo no tiene acceso directo a gran parte de los datos que se encuentran en el mismo. De esta forma, para acceder a los datos encontrados dentro del área de almacenamiento privada de las aplicaciones o para crear una disk image del dispositivo, debemos hacer lo

que se conoce popularmente como *rooting* del dispositivo.

Rooting es el proceso mediante el cual se obtiene root en un dispositivo Android. El mismo se realiza de la misma forma que otros sistemas Linux: explotando una vulnerabilidad. Tras haber obtenido root, es posible modificar el contenido del dispositivo, acceder a zonas protegidas del mismo ó incluso utilizar recursos de hardware sin las restricciones impuestas por el fabricante.

Existen tres tipos de rooting:

- **Temporal**: el dispositivo vuelve a sus condiciones normales una vez que es reiniciado.
- **Permanente**: se modifican varios archivos del firmware del dispositivo, haciendo que los privilegios se conserven al reiniciar el dispositivo.
- **Recovery mode**: se flashea una ROM customizada de la partición de recuperación, mediante la cual se inicia el sistema con privilegios de root.

Desde el punto de vista de la extracción de datos, no es necesario realizar un rooteo permanente del dispositivo, ya que un rooting temporal brinda el acceso necesario a las áreas protegidas, y los datos modificados por este proceso volverán a su estado original una vez reiniciado el dispositivo.

Algunas de las consideraciones que se deben tener en cuenta cuando se va a realizar el rooting del dispositivo son [42]:

1. Dado que el proceso de rooteo se basa en aprovechar una vulnerabilidad del dispositivo, esta vulnerabilidad depende del modelo y versión de Android del mismo. Por lo tanto, es importante conocer las caracteristicas del mismo antes de realizar el rooteo.
2. Al realizar el rooting del dispositivo se pierden las protecciones del modelo de seguridad de Android, ya que un dispositivo rooteado permite la ejecución de aplicaciones con privilegios totales.
3. Se pueden llegar a modificar áreas del dispositivo en las cuales se guardan datos del usuario.

Sin embargo, a pesar de que existen estos riesgos, el rooting es el proceso que se usa habitualmente si se quieren obtener los datos sensibles encontrados en el dispositivo.

5.5.1. Formas de obtener root

Cuando nos decidimos a obtener root en un dispositivo Android, contamos con las siguientes técnicas:

- **Verificar si tiene root**: Primero deberíamos empezar por verificar que el dispositivo no se encuentra rooteado. Para esto una forma sencilla es conectar el dispositivo a una PC y ejecutar el comando su a través del ADB (adb shell su). Para poder ejecutar este comando, es necesario que el dispositivo tenga habilitado el modo debug. Si el dispositivo está rooteado, este comando mostrará la consola del usuario root.
- **Exploits**: Existen grandes listados sobre distintos tipos de técnicas para obtener privilegios de root a través de exploits. Estos exploits pueden encontrarse en muchos sitios, siendo uno de los más populares es sitio de modders xda-developers. En el mismo se pueden encontrar exploits para la gran mayoría de dispositivos y versiones de Android, con los cuales se puede obtener privilegios de root tanto temporalmente como permanentemente.
  Tanto con esta técnica como con el comando su, es necesario que se inicie el dispositivo en su modo normal, montando la partición de datos del usuario. Al iniciar el dispositivo su modo normal, no se puede garantizar la preservación de los datos borrados (unallocated).
- **Recovery Mode**: El modo de recuperación de Android fue diseñado para realizar actualizaciones, formatear y realizar operaciones de mantenimiento sobre el dispositivo. Las funcionalidades brindadas por este modo son limitadas, y ninguna de ellas brinda la capacidad de obtener privilegios de root. Sin embargo, mediante la instalación de particiones customizadas de recuperación, se pueden obtener estos privilegios como veremos en la siguiente sección.

5.6. Alternativas a rooting

La primera alternativa es utilizar el comando `adb backup`, el cual no requiere acceso root al dispositivo pero sí que el mismo tenga activada la opción "USB Debugging" en su menú de configuración. De esta forma, este comando nos permite realizar un respaldo de los datos internos de las aplicaciones o de la tarjeta SD del dispositivo, y también se puede indicar que se extraiga el apk de las mismas.

Para completar el respaldo con éxito, se requiere intervención por parte del usuario del dispositivo móvil. Al momento de ejecutar el comando desde la consola de la PC, en la pantalla del dispositivo se muestra un diálogo, donde se solicita confirmación.

Los desarrolladores de aplicaciones cuentan con la posibilidad de deshabilitar el backup de datos de su aplicación, esto es, no permitir el respaldo de los datos que la aplicación mantiene en su región de almacenamiento interno privado. Sin embargo, observamos que la mayoría de las aplicaciones suelen tener la opción de respaldo habilitada.

La segunda alternativa es instalar una *CRMI* (Custom Recovery Mode Image). El modo de recuperación de Android fue diseñado para realizar actualizaciones, formatear y realizar operaciones de mantenimiento sobre el dispositivo. Las funcionalidades brindadas por este modo son limitadas, y ninguna de ellas brinda la capacidad de obtener privilegios de root. Sin embargo, mediante la instalación de particiones customizadas de recuperación, se pueden obtener estos privilegios. Una vez que se tiene una CRMI, se la flashea sobre la partición de booteo, y se reinicia el dispositivo. Luego de iniciado, se cuenta con privilegios de root, pudiéndose obtener todos los datos deseados. Dado que no es necesario montar la partición de datos del usuario, la copia puede realizarse sin correr el riesgo de que se alteren los datos, incluso datos borrados. Luego de realizar la extracción deseada, se debe volver a flashear la versión original de la partición de booteo [32].

Bibliografía

[1] Anders O. Flaglien, et al., *Storage and exchange formats for digital evidence*, (2011).

[2] Andrew Hoog, *Android Forensics, Chapter 4: How Data are Stored*, Elsevier, 2011.

[3] Android Developers, *Android Storage*, `http://developer.android.com/guide/topics/data/data-storage.html`, Última visita: 3 de Setiembre de 2015.

[4] Android Forums, *More information about Android partitions*, `http://androidforums.com/threads/more-information-about-android-partitions.279261/`, Última visita: 3 de Setiembre de 2015.

[5] Android Open Source Project, *Android Security Background*, `https://source.android.com/devices/tech/security/images/image00.png`, Última visita: 5 de setiembre de 2015.

[6] Bell, G.B. & Boddington, R., *Solid State Drives: The Beginning of the End for Current Practice in Digital Forensic Recovery?*

[7] Bill Anderson, *Review of Android Partition Layout*, `http://www.all-things-android.com/content/review-android-partition-layout`, Última visita: 3 de Setiembre de 2015.

[8] Brian Neil Levine, *DEX: Digital evidence provenance supporting reproducibility and comparison*, (2009).

[9] Brothers, S., *How Cell Phone Forensic Tools Actually Work - Proposed Leveling System*, 2009, Mobile Forensics World 2009. Chicago, Illinois.

[10] Cindy Murphy, *Cellular Phone Evidence: Data Extraction and Documentation*, (2010), Última visita: 3 de Setiembre de 2015.

[11] Eoghan Casey, *Digital Evidence and Computer Crime, Chapter 20*, (2015), Última visita: 3 de Setiembre de 2015.

[12] Forensic Blog, *Current Android Malware*, http://forensics.spreitzenbarth.de/android-malware/, Última visita: 3 de Setiembre de 2015.

[13] Forensic Wiki, *Dc3dd*, http://www.forensicswiki.org/wiki/Dc3dd, note=Última visita: 3 de Setiembre de 2015.

[14] ———, *Dcfldd*, http://www.forensicswiki.org/wiki/Dcfldd, Última visita: 3 de Setiembre de 2015.

[15] ———, *Dd*, http://www.forensicswiki.org/wiki/Dd, Última visita: 3 de Setiembre de 2015.

[16] ———, *Disk Image*, http://forensicswiki.org/wiki/Disk_image, Última visita: 3 de Setiembre de 2015.

[17] ———, *Raw Image*, http://forensicswiki.org/wiki/Raw_Image_Format, Última visita: 3 de Setiembre de 2015.

[18] Gartner, *Annual smartphone sales surpassed sales of feature phones for the first time in 2013*, http://www.gartner.com/newsroom/id/2665715, 2013, Última visita: 3 de Setiembre de 2015.

[19] Google, *Project Ara*, http://www.projectara.com, Última visita: 3 de Setiembre de 2015.

[20] Haroon Q Raja, *Android Partitions Explained*, http://www.addictivetips.com/mobile/android-partitions-explained-boot-system-recovery-data-cache-misc/, Última visita: 3 de Setiembre de 2015.

[21] InfoSec Institute, *SIM Card Forensics: An Introduction*, http://resources.infosecinstitute.com/sim-card-forensics-introduction/, Última visita: 3 de Setiembre de 2015.

[22] JEDEC, *Universal Flash Storage*, http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs, Última visita: 3 de Setiembre de 2015.

[23] ———, *Embedded Multi-Media Card (eMMC), Electrical Standar (5.1)*, (2012).

[24] Joachim Metz, *LibEWF Project*, https://github.com/libyal/libewf, note=Última visita: 3 de Setiembre de 2015.

[25] Joachim Metz, Robert-Jan Mora, *An Introduction To The libewf Expert Witness Library*, http://www.sleuthkit.org/informer/sleuthkit-informer-23.html#libewf, note=Última visita: 3 de Setiembre de 2015.

[26] Martin Mulazzani, *Social Network Forensics: Tapping the Data Pool of Social Networks*, https://www.sba-research.org/wp-content/uploads/publications/socialForensics_preprint.pdf, Última visita: 3 de Setiembre de 2015.

[27] Michael Cohen, Simson Garfinkel, Bradley Schat, *Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow*, (2009).

[28] Michael Harrington, *Android Google Maps Location*, https://mobileforensics.wordpress.com/2011/04/23/android-google-maps-location/, 2011, Última visita: 3 de Setiembre de 2015.

[29] MITRE, *Custom Objects*, http://cybox.readthedocs.org/en/latest/examples.html#custom-objects, Última visita: 3 de Setiembre de 2015.

[30] ———, *Cyber Observable eXpression*, http://cybox.mitre.org/documents/Cyber%20Observable%20eXpression%20(CybOX)%20Foundations%20-%20(SwA%20Forum%20Spring%202012)%20-%20Sean%20Barnum.pdf, Última visita: 3 de Setiembre de 2015.

[31] ———, *Repositorio GitHub de CybOX*, http://cyboxproject.github.io/, Última visita: 3 de Setiembre de 2015.

[32] Namheun Son, et al., *A study of user data integrity during acquisition of Android devices*, (2013).

[33] Neal Ekker et al., *Solid State Storage 101: An introduction to Solid State Storage*, (2009).

[34] NIST, *Guidelines on Mobile Device Forensics*, (2013).

[35] OASIS, *Advances Automated Cyber Threat Intelligence Sharing with STIX, TAXII, CybOX*, https://www.oasis-open.org/news/pr/oasis-advances-automated-cyber-threat-intelligence-sharing-with-stix-taxii-cybox, Última visita: 3 de Setiembre de 2015.

[36] OpenSignal, *Android fragmentation visualized*, http://opensignal.com/reports/fragmentation-2013/, 2013, Última visita: 3 de Setiembre de 2015.

[37] Philip Turner, *Unification of digital evidence from disparate sources (Digital Evidence Bags)*, (2005).

[38] S. Garfinkel, et al., *Advanced Forensics Format: An open, extensible format for disk imaging*, http://cs.harvard.edu/malan/publications/aff.pdf, Última visita: 3 de Setiembre de 2015.

[39] SD Association, *An introduction to SD Cards*, https://www.sdcard.org/developers/overview/index.html, Última visita: 3 de Setiembre de 2015.

[40] Teck Sung Yap, Hong Tat Ewe, *A Mobile Phone Malicious Software Detection Model with Behavior Checker*, (2005).

[41] TheMacObserver, *Proposed EU Law Could Force Apple to Change iPhone Charging Connector*, http://www.macobserver.com/tmo/article/proposed-eu-law-could-force-apple-to-change-iphone-charging-connector, Última visita: 3 de Setiembre de 2015.

[42] Timothy Vidas, et al., *Towards a General Collection Methodology for Android Devices*, (2011).

[43] TutorialsPoint.

[44] UFSA, *UFS 2.0 will see rapid adoption*, http://universalflash.org/2014/02/ufs-2-0-will-see-rapid-adoption/, Última visita: 3 de Setiembre de 2015.

[45] W. Alink, *XIRAF, XML-based indexing and querying for digital forensics*, (2006).

[46] Wikipedia, *Android version history*, https://en.wikipedia.org/wiki/Android_version_history, Última visita: 3 de Setiembre de 2015.

[47] _____, *Disk Image*, https://en.wikipedia.org/wiki/Disk_image, Última visita: 3 de Setiembre de 2015.

[48] _____, *History of iOS*, https://en.wikipedia.org/wiki/IOS_version_history, Última visita: 3 de Setiembre de 2015.

[49] _____, *History of Microsoft Windows versions*, https://en.wikipedia.org/wiki/History_of_Microsoft_Windows, Última visita: 3 de Setiembre de 2015.

[50] _____, *iBeacon*, http://en.wikipedia.org/wiki/IBeacon, Última visita: 3 de Setiembre de 2015.

[51] _____, *Servicio de mensajes cortos*, http://es.wikipedia.org/wiki/Servicio_de_mensajes_cortos, Última visita: 3 de Setiembre de 2015.

[52] _____, *Wi-Fi positioning system*, http://en.wikipedia.org/wiki/Wi-Fi_positioning_system, Última visita: 3 de Setiembre de 2015.

[53] XDA Developers, *Backup And Restore Your EFS Folder On Samsung Devices*, http://www.xda-developers.com/backup-and-restore-your-efs-folder-on-samsung-devices/, Última visita: 3 de Setiembre de 2015.

[54] Yi Sun, *Geo-Location Forensics on Mobile Devices*, (2011).

[55] ZDNet, *How Google–and everyone else–gets Wi-Fi location data*, http://www.zdnet.com/blog/networking/how-google-and-everyone-else-gets-wi-fi-location-data/1664, Última visita: 3 de Setiembre de 2015.