

Binary Analysis Tool User and Developer Manual

Armijn Hemel – Tjaldur Software Governance Solutions

January 30, 2012

1 Introducing the Binary Analysis Tool

The Binary Analysis Tool (BAT) is a tool that can help developers and companies check binary files. Its primary application is for Open Source software license compliance, with a special focus on supply chain management in consumer electronics.

BAT consists of several scripts written in Python. The most important program is the scanner for binary objects to detect the presence of Open Source software inside binaries. There are also other programs to help with specific tasks, such as verifying if configurations for a given BusyBox binary match with the configuration in source code. Also included is a very experimental program to help scan Linux kernel images and a script to derive a Linux kernel configuration from a Linux kernel image.

2 Installing the Binary Analysis Tool

2.1 Software requirements

To run BAT a recent Linux distribution is needed. Development is (currently) done on Fedora 14 and Fedora 15, so those platforms are likely to work best. Other platforms (Debian 6, Ubuntu 10.10 and later) are tested with, but only to build binary releases.

If the latest version from version control is used it is important to look at the file `setup.cfg` to get a list of the dependencies that should be met.

2.2 Hardware requirements

The tools in the Binary Analysis Tool can be quite resource intensive. Most of the tools seem to be memorybound and I/O-bound, and it makes more sense to invest in more memory and faster disks than in raw CPU power, although using more cores definitely will speed up the scanning process.

2.2.1 Installing on Fedora

To install on Fedora three packages are needed: `bat-extratools`, `bat-extratools-java` and `bat`. These can be downloaded from the website of the binary analysis tool in both prebuilt versions and as a source RPM file. When installing the three files there should be a list of dependencies that should be installed to let BAT work successfully. Some of these packages are not in Fedora by default but need to be installed through extra repositories, such as RPMfusion.

2.2.2 Installing on Debian and Ubuntu

To install on Debian and Ubuntu three packages are needed: `bat-extratools`, `bat-extratools-java` and `bat`. These can be downloaded from the website of the binary analysis tool in both prebuilt versions. When installing the three files there should be a list of dependencies that should be installed to let BAT work successfully. Some of these packages are not in Debian by default but need to be installed by enabling extra repositories such as Debian `non-free`.

3 Using the programs in the Binary Analysis Tool

BAT consists of several programs and a few helper scripts (not meant to be used directly).

The programs in the Binary Analysis Tool are `bruteforce.py`, `busybox-compare-configs.py` and `busybox.py`

3.1 bruteforce.py

The main program in BAT is `bruteforce.py`. As the name suggests uses a brute force approach for scanning a binary. It assumes no prior knowledge of how a binary is constructed or what is inside the binary. Instead it tries to determine what is inside by applying several methods, such as looking for known identifiers of file systems and compressed files and running external tools to find contents in the binary.

After successful identification it unpacks compressed files, media files and file systems recursively. Finally several individual scans are applied to each extracted file, ranging from determining dynamically linked libraries to an elaborate ranking method to guess which sources were used to build a program.

3.1.1 Running bruteforce.py

The `bruteforce.py` tool uses a configuration file that is in Windows INI format. The main configuration directive is `batconfig`, which currently only allows setting a single setting, namely whether or not multiple CPUs (or cores) should be used during scanning. The default configuration as shipped in the official BAT distribution is to not use multiple CPUs:

```
[batconfig]
multiprocessing = no
```

By setting this to `yes` the program will start an extra process per CPU that is available. In most cases it is completely safe to use multiprocessing, but there is one important exception: if the ranking method (described later) is used, without pregenerating the proper caching database (or purchasing one) it is unsafe to use multiprocessing. If ranking is used and there is a fully generated database it is completely safe and even strongly advised to use multiprocessing, since it vastly speeds up running the program.

3.1.2 Marker search

The first action performed by `bruteforce.py` is to scan a file for known markers from compressed files and file systems. Which markers are searched for depends on the contents of the configuration file (specifically the `magic` attribute). These markers are subsequently used to choose which other methods, such as unpackers, to run.

3.1.3 Pre-run checks

Before files are unpacked they are briefly inspected and tagged, so methods that are applied later do not unnecessarily have to process files, reducing run time and false positives.

For example, files that only contain text are tagged as `text`, and the rest as `binary` (this depends on the implementation of Python. Python 2 only considers (by default) ASCII to be valid text). Methods that only work on binaries can ignore anything that has been tagged with `text`.

There are similar checks to determine valid XML and images, plus there are a few example methods for more sophisticated checks, such as detecting `gzip` and `bzip2` compression, which are disabled by default.

3.1.4 Unpackers

Unpackers can be recognized in the configuration because their type is set to `unpack`, for example:

```
[jffs2]
type    = unpack
module  = bat.fwunpack
method  = searchUnpackJffs2
priority = 1
```

In BAT 6.0 the following file systems, compressed files and media files can be unpacked or extracted:

- file systems: `cramfs`, `ext2/ext3/ext4`, `ISO9660`, `JFFS2`, `SquashFS` (several variants), `UBIFS` (not on Debian/Ubuntu), `YAFFS2` (specific variants)
- compressed files and executable formats: `7z`, `ar`, `ARJ`, `BASE64`, `BZIP2`, compressed `Flash`, `CAB`, `CPIO`, `EXE` (specific compression methods only) `GZIP`, `InstallShield` (old versions), `LRZIP`, `LZIP`, `LZMA`, `LZO`, `RAR`, `RPM`, serialized `Java`, `TAR`, `UPX`, `XZ`, `ZIP`
- media files: `GIF`, `ICO`, `PDF`, `PNG`

Unpacking differs per file type. Most files use one or more identifiers that can be searched for in a binary blob. Using this information it is possible to carve out the right parts of a binary blob and verify if it indeed contains a compressed file, media file or file system.

There is not always an identifier that can be searched for. The `YAFFS2` file system layout for example is dependent on the hardware specifics of the underlying flash chip. Without knowing these specifics it is not possible to unpack `YAFFS2` file systems.

Other files (such as ARJ and ICO files) have a very generic identifier, so there are a lot of false positives. This causes a big increase in runtime. These two unpackers are therefore disabled by default.

If unpacking is successful a directory with unpacked files that can be scanned further is returned, and, if available, some meta information to avoid duplicate scanning (blacklisting information and tags).

3.1.5 Leaf scans

Leaf scans, also called program scans, are scans that are run on every single file after unpacking, including files that contained files that were found and extracted by unpackers.

Leaf/program scans can be recognized in the configuration because their type is set to `program`, for example:

```
[iptables]
type      = program
module    = bat.checks
method    = searchIptables
```

The current program scans that are available in BAT are:

- fast string searches: `dproxy`, `ez-ipupdate`, `iptables`, `iproute`, `libusb`, Linux kernel, `loadlin`, `RedBoot`, `U-Boot`, `vsftpd`, `wireless-tools`, `wpa-supPLICANT`
- advanced search mode using ranking and a large database
- BusyBox version number
- dynamic library dependencies (ELF files only)
- Linux kernel module license (Linux kernel modules only)

The fast string searches are meant for quick sweep scanning only. They have their limits, can report false positives or miss finding a program. They should only be used to signal that further inspection is necessary. For a thorough investigation the advanced search mode should be used.

3.1.6 Post-run methods

Since BAT 6.0 there is the possibility to add methods that are run after all the regular work has been performed, or “post-run”. These methods should not alter the scan results in any way, but just use the information from the scanning process. A typical use case would be to present the data in a nicer to use format than the standard report, or to use more external data sources.

While there are no post-run methods in BAT by default it is fairly easy to write them.

3.1.7 Enabling and disabling scans and methods

The standard configuration file enables most of the scans and methods implemented in BAT by default. To disable a scan it can be outcommented in the file (by starting the line with the `#` character), or by removing it from the configuration file.

3.1.8 Interpreting the results

`bruteforce.py` outputs its results in XML format on standard output. After redirecting the output to a file it is possible to look at this file with a commandline tool such as `xml_pp` or a webbrowser such as Mozilla Firefox.

Since XML is not meant for human consumption a graphical user interface is planned that wraps around BAT that will make it easier to interpret the results in a more interactive way.

The XML file starts with metadata, such as:

- date, plus time of the scan (local time of the computer, in UTC)
- name of the file
- SHA256 cryptographic checksum of the file, uniquely identifying it
- size of the file
- filetype as determined by `file` on a Linux system
- relative path inside the unpacked system, plus the absolute path inside the file system, which is useful for later analysis

If any of the scans were successful the results of these scans can be found in the element `scans`.

For each successful unpack action the following attributes are reported:

- name of the scan (corresponding to the name of the scan in the configuration file)
- offset in the parent file of the compressed file, file system or media file

3.2 `busybox.py` and `busybox-compare-configs.py`

Two other tools in BAT are `busybox-compare-configs.py` and `busybox.py` (in the subdirectory `bat`). These two tools are specifically used to analyze BusyBox binaries. BusyBox is in widespread use on embedded devices and the license on BusyBox is actively enforced in court.

BusyBox binaries on embedded machines often have different configurations, depending on the needs of the manufacturer. Since providing the correct configuration is one of the requirements for license compliance it is important to be able to determine the configuration of a BusyBox binary and verify that there is a corresponding configuration file in the source code release.

The BusyBox processing tools in BAT try to extract the most likely configuration from the binary and print it in the right format (which depends on the BusyBox version).

`busybox.py` is used to extract the configuration from a binary. Afterwards `busybox-compare-configs.py` can be used to compare the extracted configuration with a vendor supplied configuration.

3.2.1 Extracting a configuration from BusyBox

Extracting a configuration from a BusyBox executable is done using `busybox.py` which can be found in the `bat` directory. It needs two commandline parameters: the path to the binary and the path to a directory containing a directory `configs` which has files containing mappings from BusyBox applet names to BusyBox configuration directives. By default this value is hardcoded as `/etc/bat`, but this might change in the future.

```
python bat/busybox.py -b /path/to/busybox/binary -c
/path/to/pre/extracted/configs > /path/to/saved/config
```

This command will save the configuration to a file, which can be used as an input to `busybox-compare-configs.py`.

3.2.2 Comparing two configurations

After extracting the configuration the extracted configuration can be compared to another configuration, for example a configuration as supplied by a vendor:

```
python busybox-compare-configs.py -e /path/to/saved/config
-f /path/to/vendor/configuration -n $version
```

3.3 `extractkernelstrings.py`, `extractkernelconfig.py` and `findkernelstrings.py`

The three programs `extractkernelstrings.py`, `extractkernelconfig.py` and `findkernelstrings.py` are very experimental tools to derive a configuration from a Linux kernel image. They are far from mature and are not yet ready for production use.

Once they are ready for production use a description for the scripts will be included in this manual. A description of the technical uses that need to be solved, plus a motivation on why and how to solve them can be found in the appendix.

4 Binary Analysis Tool extratools collection

To help with unpacking non-standard file systems, or standard file systems for which there are no tools readily available on Fedora or Ubuntu there is also a collection of tools that can be used by BAT to unpack more file systems. These tools are not part of the standard distribution, but have to be installed separately. They are governed by different license conditions than the core BAT distribution.

Currently the collection consists of:

- modified version of `cramfsck` that enables unpacking `cramfs` file systems
- unmodified version of `unyaffs` that enables unpacking for some (but not all) `YAFFS2` file systems.
- various versions of `unsquashfs` that enable unpacking variants of `SquashFS`. These versions have either been lifted from vendor SDKs, the `OpenWrt` project, or upstream `SquashFS` project.

- two Java projects: `jdserialize` and `ddex`, to help respectively with unpacking serialized Java files and scanning binary files from the Dalvik VM (Android).

The collection is split in two packages: `bat-extratools-java` contains the two Java packages, the `bat-extratools` package contains the rest.

A bruteforce.py internals

The `bruteforce.py` program processed binaries: it unpacks files by running unpackers, it runs scans on individual files with program/leaf scans, processes and prettyprints output and performs cleanups, if necessary.

`bruteforce.py` was written with extensibility in mind: new file systems or variants of old ones tend to appear regularly (for example: there are at least 5 or more versions of SquashFS with LZMA compression out there).

A.1 Pre-run methods

Pre-run methods check and tag files, so the files can be ignored by later methods and scans. While tagging is not exclusive to pre-run methods it is their main purpose.

A.1.1 Writing a pre-run method

Pre-run methods have a strict interface:

```
def prerunMethod(filename, tempdir=None, tags=[],
                 offsets={}, envvars=None):
    newtags = []
    newtags.append('helloworld')
    return newtags
```

- `filename` is the absolute path of the file that needs to be tagged
- `tempdir` is the (possibly) empty name of a directory where the file is. This is currently unused and might be removed in the future.
- `tags` is the set of tags that have already been defined for the file.
- `offsets` is the set of offsets that have been found for the file
- `envvars` is an optionally empty set of environment variables that can be used to pass extra information to the pre-run method.

Return values are:

- a list containing tags

A.2 Unpackers

Unpackers are responsible for recursively unpacking binaries until they can't be unpacked any further.

A.2.1 Writing an unpacker

The unpackers have a strict interface:

```
def unpackScan(filename, tempdir=None, blacklist=[], offsets={}):  
    ## code goes here
```

The last three parameters are optional, but in practice they are always passed by the top level script.

- `tempdir` is the directory into which files and directories for unpacking should be created. If it is `None` a new temporary directory should be created.
- `blacklist` is a list of byte ranges that should not be scanned. If the current scan needs to blacklist a byte range it should add it to this list after finishing a scan.
- `offsets` is a dictionary containing a mapping from an identifier to a list of offsets in the file where these identifiers can be found. This list is filled by the scan `genericMarker` which always runs before anything else.

Return values are:

- the name of a directory, containing files that were unpacked.
- the blacklist, possibly appended with new values
- a list of tags, in case any tags were added, or an empty list

Most scans have been split in two parts: one part is for searching the identifiers, correctly setting up temporary directories and collecting results. The other part is doing the actual unpacking of the data and verification.

The idea behind this split is that sometimes functionality is shared between two scans. For example, `unpackCpio` is used by both `searchUnpackCpio` and `unpackRPM`.

A.2.2 Adding an identifier

Identifiers for new file systems and compressed files are, if available, added to `fsmagic.py` in the directory `bat`. These identifiers will be available in the `offsets` parameter that is passed to a scan, if any were found.

Good sources to find identifiers are `/usr/share/magic`, documentation for file systems or compressed files, or the output of `hexdump -C`.

A.2.3 Blacklisting and priorities

In BAT blacklists are used to prevent some scans from running on a particular byte range, because other scans have already covered these bytes, or will cover them.

The most obvious example is the `ext2` file system: in a normal setup (no encryption) it is trivial to see the content of all the individual files when an `ext2` file system image is opened. This is because this file system is mostly a concatenation of the data, with some meta data associated with the files in the file system.

If another compressed file is in the `ext2` file system it could be that it will be picked up by BAT twice: once it will be detected inside the `ext2` file system and once after the file system has been unpacked by the `ext2` file system unpacker.

Other examples are:

- `cpio` (files are concatenated with a header and a trailer)
- `tar` (files are concatenated with some meta data)
- `RPM` (files are in a compressed archive with some meta data)
- `ar` and `DEB`
- some flavours of `cramfs`
- `ubifs`

To avoid duplicate scanning and false positives it is therefore necessary to prevent other scans from running on the byte range already covered by one of these files.

In BAT this is achieved by using blacklists. All unpackers have a parameter called `blacklist` which is consulted everytime a file is unpacked. If a file system offset is in a blacklist the scan could use the next offset, or skip scanning the entire file, depending on the scan.

The blacklist is set for every file individually and is initially empty. If a scan is successful it adds a byte range to the blacklist. Subsequent scans will skip the byte range added by the scan.

The scans are run in a particular order to make the best use of blacklists. The order of scans is determined by the `priority` parameter in the configuration file. The file systems and concatenated files mentioned above have a higher priority and are scanned earlier than other scans that could also give a match. It is not a fool proof system, but it seems to work well enough.

A.3 Program/leaf scans

After everything has been unpacked each file, including the files from which other files were carved, will be scanned again by the program, or leaf, scans.

A.3.1 Writing a leaf scan

The program/leaf scans have a simple interface. There are two parameters passed down, namely the absolute path of the file, plus an optional blacklist with byte ranges that should not be scanned. For example:

```
def programScan(path, blacklist=[]):  
    ## code goes here
```

There are no restrictions on the return values of the program/leaf scan, except when nothing could be found. The result values are:

- `None` if nothing can be found
- simple values (booleans, strings)
- custom data structure, but a custom pretty printer has to be added

There is no restriction on the code that is run and basically anything can be done. In BAT there are for example checks that invoke other programs to discover dynamically linked libraries, find the license of a kernel module using `modinfo` or simple checks for the presence of strings in the binary that indicate the use of certain software.

The simplest methods are the ones that scan for hardcoded strings. These strings are frequently found just in the package for which the check is written for. For example, these strings can often be found in copies of the `iptables` program and the related `libiptc` library:

```
markerStrings =
    [ 'iptables who? (do you need to insmod?)'
      , 'Will be implemented real soon. I promise ;)'
      , 'can\'t initialize iptables table '%s\': %s'
    ]
```

Although this is very fast, this method has some drawbacks:

- a program sometimes does not have these exact strings embedded in the binary
- this method will only find the strings that are hardcoded and not any other significant strings
- if another package includes the string, there is a false positive

The quick checks should therefore only be used as an indication that further inspection of the binary is needed. A much better method is the ranking method that is also available in BAT, but which is not as fast.

A.3.2 Pretty printing for leaf scans

Pretty printing for unpackers is standardized but for leaf scans there is more flexibility. This is needed because in some cases the result as returned by the leaf scan needs post processing.

A pretty printer can be defined in the configuration through the directive `xmloutput`. The pretty printer should be in the same module as the scanning method defined in the same section. The pretty printer has two parameters: a Python datastructure as returned by the scanner (this differs per scan) and a XML root element, needed to create new XML nodes. The method is expected to return a XML node in case of success, or `None` in case of failure.

If no pretty printer is defined the value as returned by the scan will be used as the content of result tag.

A.4 Post-run methods

Post-run methods don't change the result of the whole scanning process, but only use the data from the process. For example prettyprinting a fancy report (more advanced than the standard XML report) would be a typical post-run method.

A.4.1 Writing a post-run method

Post-run methods have a strict interface:

```
def postrunHelloWorld(filename, unpackreports, leafreports, envvars={}):
    print "Hello World"
```

- `filename` is the absolute path of the scanned file, after unpacking.
- `unpackreports` are the reports of the unpackers for the file
- `leafreports` are the reports of the unpackers for the file
- `envvars` is an optional dictionary of environment variables

Since the post-run methods don't change the result in any way, but just have side effects there is no need to return anything.

B Building binary packages of the Binary Analysis Tool

If you want to install BAT through the package manager of your distribution you might first need to generate packages for it if none exist. For BAT there is currently support to build packages for RPM-based systems and for DEB-based systems.

B.1 Building packages for RPM based systems

B.1.1 Building `bat`

Building the `bat` package is fairly straightforward.

1. Make a fresh export of BAT from Subversion
2. run the command: `python setup.py bdist_rpm`

This will create an RPM file and an SRPM file. If you need to install BAT on other versions of Fedora or on other RPM based distributions you can simply rebuild the SRPM using `rpmbuild --rebuild`.

B.1.2 Building `bat-extratools` and `bat-extratools-java`

Building packages for `bat-extratools` and `bat-extratools-java` is unfortunately a bit more elaborate.

1. make a fresh export of the Subversion repository
2. change the names of `bat-extratools` and the `bat-extratools-java` directories to contain the version name of the release (for example `bat-extratools-6.0`). Make a `tar.gz` archive of the directory: `tar zcf bat-extratools-6.0.tar.gz bat-extratools-6.0`
3. run `rpmbuild` to create binary packages: `rpmbuild -ta bat-extratools-6.0.tar.gz`

B.2 Building packages for DEB based systems

B.2.1 Building bat

The Debian scripts were written according to the documentation for `debhelper` found at <https://wiki.ubuntu.com/PackagingGuide/Python>.

Package building and testing is done on Ubuntu 10.10 and Debian 6.

To build a `.deb` package do an export of the Subversion repository first. Change to the directory `src` and type: `debuild -uc -us` to build the package.

The build process might complain about not being able to find the original sources. In our experience it is safe to ignore this. The command will build a `.deb` package which can be installed with `dpkg -i`.

B.2.2 Building bat-extratools and bat-extratools-java

To build a `.deb` package do an export of the Subversion repository first. Change to the correct directories (`bat-extratools` and `bat-extratools-java` and type: `debuild -uc -us` to build the packages.

C Binary Analysis Tool knowledgebase

BAT comes with a mechanism to use a database backend. The default version of BAT only unpacks file systems and compressed files and runs a few simple checks on the leaf nodes of the unpacking process.

In the paper “Finding Software License Violations Through Binary Code Clone Detection” by Hemel et. al. (ACM 978-1-4503-0574-7/11/05), presented at the Mining Software Repositories 2011 conference, a method to use a database with strings extracted from source code was described. This functionality is available in the ranking module in the file `ranking.py`. This code is not enabled by default, but it has to be explicitly enabled in the configuration for `bruteforce.py`.

To give good results the database that is used needs to be populated with as many packages as possible, from a cross cut of all of open source software, to prevent bias towards certain packages: if you only would have BusyBox in your database, everything would look like BusyBox.

If you don't want to spend much time on downloading and processing packages, please contact Tjaldur Software Governance Solutions for purchasing a copy of a fully prepared database at info@tjaldur.nl.

C.1 Crawling FTP mirrors

In the directory `crawlers` a sample crawler is given for downloading source packages from the GNU project. The code is fairly straightforward. The script (written in Python) tries to intelligently download packages from a GNU FTP mirror, skipping packages that are in a blacklist (either parts of the filename, whole directories, or extensions) or that have already been downloaded in the past.

The default configuration points to a mirror in the Netherlands. It is advised to update the location of the GNU mirror in the configuration file. Specifying a local mirror will result in faster downloads. It will also not put an unnecessary strain on the FTP mirror mentioned in the default configuration file.

C.2 Generating the package list

The code and license extractor wants a description file of which packages to process. This file is hardcoded to `LIST` relative to the directory that contains all source archives. The reason there is a specific file is that some packages do not follow a consistent naming scheme. By using this extra file we can cleanup names and make sure that source code archives are recognized correctly.

The file contains four values per line:

- name
- version
- archivename
- origin (or “unknown” if not specified)

separated by whitespace (spaces or tabs). An example would look like this:

```
amarok 2.3.2 amarok-2.3.2.tar.bz2 kde
```

This line says that the package is `amarok`, the version number is `2.3.2`, the filename is `amarok-2.3.2.tar.bz2` and the file was downloaded from the KDE project.

There is a helper script (`generatelist.py`) to help generate the file. It can be invoked as follows:

```
python generatelist.py -f /path/to/directory/with/sources
```

The output is printed on standard output, so you want to redirect it to a file called `LIST` (as expected by the string extraction script) and optionally sorting it first:

```
python generatelist.py -f /path/to/directory/with/sources | sort >
/path/to/directory/with/sources/LIST
```

`generatelist.py` tries to determine the name of the package by splitting the file name on the right on a `-` (dash) character. This is not always done correctly because a package uses multiple dashes, or because it does not contain a dash. In the latter case an error will be printed on standard error, informing you that a file could not be added to the list of packages and it should be added manually.

It is advised to manually inspect the file after generating it to ensure the correctness of the package names. Packages can have been renamed for a number of reasons:

- upstream projects decided to use a new name for archives (AbiWord archives for example were renamed from `abi-$VERSION.tar.gz` (used for early versions) to `abiword-$VERSION.tar.gz`).
- a distribution has renamed packages to avoid clashes during installation and allow different versions to be installed next to eachother.

In these cases you need to change the names of the packages, otherwise different versions of the same package will be recorded in the database as different packages, which will confuse the rating algorithm and cause it to give suboptimal results.

C.3 Running the extraction program

The program to extract strings from sourcecode is `batchextractprogramstrings.py`. It parses the file generated by `generatelist.py`, unpacks the files (currently only `tar.gz` and `tar.bz2` files, plus variants are supported) and scans each individual source code file (at the moment just C/C++ files, assembler files, C# files, Java/Scala files and ActionScript files) for strings and, if enabled, licenses using Ninka and FOSSology.

`batchextractprogramstrings.py` can be invoked as follows:

```
python batchextractprogramstrings.py -f
    /path/to/directory/with/files -d /path/to/database
```

C.4 License scanning

`batchextractprogramstrings.py` has a few commandline options. The most important is whether or not to also extract licenses from the source code files. License extraction is done using the Ninka license scanner, with support for the Nomos license scanner from FOSSology being added in the near future. This option is disabled by default for a few reasons:

- scanning licenses adds a significant performance penalty to scanning the code (about 1100% when scanning `uClibc 0.9.30.1`).
- there is no code in the BAT 6.0 that makes use of this information.

If you want to enable license scanning, you will have to install Ninka first and change a few hardcoded paths in `batchextractprogramstrings.py` that point to the main Ninka script.

C.5 Setting the database

Since BAT 6.0 the location of the strings database and the caching databases can be set in the configuration file for `bruteforce.py`:

```
[ranking]
type      = program
module    = bat.ranking
method    = searchGeneric
xmloutput = xmlprettyprint
envvars   = BAT_SQLITE_AVG=/tmp/avg:BAT_SQLITE_STRINGSCACHE=/tmp/stringscache
           :BAT_SQLITE_DB=/tmp/master
```

The caching databases are `BAT_SQLITE_AVG` and `BAT_SQLITE_STRINGSCACHE`. The strings database is `BAT_SQLITE_DB`.

Fallback values for these databases have been hardcoded in `ranking.py`. To change these values, change the following lines:

```
conn = sqlite3.connect(scanenv.get('BAT_SQLITE_DB', '/tmp/master'))
avgdb = scanenv.get('BAT_SQLITE_AVG', '/tmp/avg')
stringscache = scanenv.get('BAT_SQLITE_STRINGSCACHE', '/tmp/stringscache')
```

C.6 Enabling the ranking scan

The ranking scan is configured in the configuration file. In the configuration shipped with BAT it is disabled by default. By uncommenting the entry for the scan it can be enabled.

Since BAT 6.0 a distinction is made in the ranking scan between programs written in different language families. When the strings are extracted from source code the language family is also recorded with the string. For example: C and C++ source code and header files, as well as QML and assembler files are recorded as C. Similarly Java, Scala, Groovy and JSP are recorded as Java, and so on.

When a binary is scanned the ranking scan first tries to classify it. Based on the classification only strings from a certain language family are used. For example, when scanning an ELF executable, only strings from C language files are selected from the database. Others, such as strings extracted from Java files, are ignored. Vice versa, for Java executables everything but Java will be ignored.

The reason for this is that there is usually little code cloning between C and Java files, or Java and C# files (except of course when interpreters and virtual machines are embedded, or other language bridges are used). Strings in a Java program are unlikely to originate from a file in a C program. Using all strings from all programs, irrespective of the language they were written in, therefore is likely to result in incorrect matching behaviour.

C.6.1 Interpreting the results

The results of the scan can be found in the element `<ranking>`. This element contains:

- number of lines that were extracted from the binary
- number of lines that could be matched exactly with an entry in the database
- result per package which are a possible match

Per package the following is reported:

- name of the package
- all unique matches (strings that can only be found in this package)
- relative ranking
- percentage of the total score

For example, take the results of a run on a BusyBox binary:

```
<ranking>
  <matchedlines>1314</matchedlines>
  <extractedlines>3147</extractedlines>
  <package>
    <name>busybox</name>
    <uniquematches>
      <unique>%d heads, %d sectors/track, %d cylinders</unique>
      ...
    </uniquematches>
```

```
<rank>1</rank>
<percentage>98.3386895181</percentage>
</package>
...
</ranking>
```

About 98% of the total score was for BusyBox, so it is a clear match. In programs where two or more packages are embedded percentages will be distributed in a different, more uniform, way.

C.7 Database design

The database currently has 5 tables:

- `processed`
- `processed_file`
- `extracted_file`
- `licenses`
- `ninkacomment`

C.7.1 `processed` table

This table is to keep track of which versions of which packages were scanned. Its only purpose is to avoid scanning packages multiple times. It is not actively used in the ranking code.

It has the following fields:

- name of the package
- version of the package
- name of the archive that was processed
- sha256 of the archive that was processed

C.7.2 `processed_file` table

This table contains information about individual source code files that were scanned.

It has the following fields:

- name of the package the file is from
- version of the package the file is from
- relative path inside the source code archive
- sha256 of the file

C.7.3 processed table

This table stores the individual strings that were extracted from files and that could possibly end up in binaries.

It has the following fields:

- string that was extracted
- sha256 of file the string was extracted from
- language of the source code file
- line number where the extracted string can be found in the source code file, or 0 if unknown. This information is determined using `xgettext`.

C.7.4 licenses table

This table stores the licenses that were extracted from files using a source code scanner, like Ninka or FOSSology. If a file has more than one licenses there will be multiple rows for a file. It has these fields:

- sha256 checksum of file
- license as found by the scanner
- scanner name. Currently only Ninka and FOSSology are used. Of course, the scanner could also be a person doing a manual review.
- version of scanner. This is useful if there is for example a bug in a scanner, or to compare results from various versions.

C.7.5 ninkacomment table

This is a helper table that keeps track of which licenses were associated with a file header as extracted by Ninka. Ninka determines licenses by looking at just the header of the file. Since the headers in a single program are often similar it is possible to get some speed improvements by first determining if a header has already been scanned.

It has the following fields:

- sha256 of the header as extracted by Ninka. Ninka stores these in separate files.
- license as found by Ninka
- name of the scanner (hardcoded to 'Ninka')
- version of Ninka

D BusyBox script internals

The BusyBox processing scripts look simple, but behind the internals are a bit hairy. Especially extracting the correct configuration is not trivial.

D.1 Detecting BusyBox

Detecting if a binary is indeed BusyBox is trivial, since in a BusyBox binary there are almost always clear indication strings if BusyBox is used (unless they it was specifically altered to hide the use of BusyBox).

A significant set of strings to look for is:

```
BusyBox is a multi-call binary that combines many common Unix
utilities into a single executable. Most people will create a
link to busybox for each function they wish to use and BusyBox
will act like whatever it was invoked as!
```

Another clear indicator is a BusyBox version string, for example:

```
BusyBox v1.15.2 (2009-12-03 00:14:42 CET)
```

As an exception a BusyBox binary configured to include just a single applet will not contain the marker strings, or the BusyBox version string. In such a case a different detection mechanism will have to be used, for example the ranking code as used in `bruteforce.py`, although this will only be necessary in a very small percentage of cases, since the vast majority of BusyBox instances include more than one applet.

D.2 BusyBox version strings

The BusyBox version strings have remained fairly consistent over the years:

```
BusyBox v1.00-rc2 (2006.09.14-03:08+0000) multi-call binary
BusyBox v1.1.3 (2009.09.11-12:49+0000) multi-call binary
BusyBox v1.15.2 (2009-12-03 00:14:42 CET)
```

The time stamps in the version string are irrelevant, since they are generated during build time and are not hardcoded in the source code.

Extracting version information from the BusyBox binary is not difficult. Using regular expression it is possible to look for `BusyBox v` which indicates the start of a BusyBox version string. The version number can be found immediately following this substring until `(` (including leading space) is found.

Apart from reporting, the BusyBox version number is also used for other things, such as determining the right configuration format and accessing a knowledgebase of known applet names extracted from the standard BusyBox releases from `busybox.net`.

D.3 BusyBox configuration format

During the compilation of BusyBox a configuration file is used to determine which functionality will be included in the binary. The format of this configuration file has changed a few times over the years. Early versions used a simple header format file, with GNU C/C++ style defines. Later versions, starting 1.00pre1, moved to Kbuild, the same configuration system as used by for example the Linux kernel or OpenWrt. This format is still in use today (BusyBox 1.19.2 being the latest version at the time of writing).

Each configuration directive determines whether or not a certain piece of source code will be compiled and up in the BusyBox binary. This source code can either be a full applet, or just a piece of functionality that merely extends an existing applet.

D.4 Extracting a configuration from a BusyBox binary

Extracting the BusyBox configuration from a binary is not entirely trivial. There are a few methods which can be used:

1. run `busybox` (on a device, or inside a sandbox) and see what functionality is reported. This is probably the most accurate method, but also the hardest, since it requires access to a device, or a sandbox that has been properly set up, with all the right dependencies, and so on.

When running `busybox` without any arguments, or with the `--help` parameter it will output a list of functions that are defined inside the binary:

Currently defined functions:

```
ar, cal, cpio, dpkg, dpkg-deb, gunzip, zcat
```

These can be mapped to a configuration, using information extracted from BusyBox source code about which applets map to which configuration option.

2. extract the configuration from the binary by searching for known applet names in the firmware. The end result is the same as a previous step, but possibly with less accuracy in some cases but it is the only feasible solution when you only have a binary.

The BusyBox binary has a string embedded for every applet that is included. This is the string that is printed out if `--help` is given as a parameter to an invocation of `busybox`.

Using information about the configuration extracted from BusyBox source code these strings can be mapped to a configuration directive and a possible configuration can be reconstructed.

Depending on how the binary was compiled this can be trivial, or quite hard.

D.4.1 Binaries linked with uClibc

In binaries that link against uClibc (a particular C library) the name of the main function of the applet is sometimes (but not always) included in the `busybox` binary as follows (a good way is to run `strings` on the binary and look at the output).

```
wget_main
```

This string maps to the name of the main function for the `wget` applet (`networking/wget.c`):

```
int wget_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE;
```

The BusyBox authors are pretty strict in their naming and usually have a configuration directive in the a specific format (`CONFIG-$appletname`) in the Makefile, like:

```
lib-$(CONFIG_WGET) += wget.o
```

(example taken from `networking/Kbuild` in BusyBox 1.15.2). There are cases where the format could be slightly different.

D.4.2 Binaries linked with glibc & uClibc exceptions

Sometimes the method described in the previous section does not work for binaries that are linked with uClibc. It also does not work with binaries compiled with glibc.

If the binary is unstripped and the binary still contains symbol information it is possible to extract the right information using `readelf` (part of GNU binutils) in a similar fashion as the earlier described method.

In case there is no information available it is still possible to search inside the binary for the applet names. Because most instances of BusyBox that are installed on devices have not been modified the list of applets in the stock version of BusyBox serves as an excellent starting point.

The list as printed by `busybox` if the `--help` parameter is given is embedded in the binary. The applet names are alphabetically sorted and separated by NUL characters.

By searching for this list and splitting it accordingly it is possible to get the list of all applets that are defined. The only caveats are that a new applet that was added appears alphabetically before any of the applets that can be recognized using a list of applet names extracted from the source code, or it appears alphabetically after the last one that can be recognized.

The current code is suboptimal and will be rewritten in the near future.

D.5 Pretty printing a configuration

Pretty printing a configuration is fairly straightforward, but there are a few cases where it is hard to make a good guess:

1. aliases
2. functionality that is added to an applet, depending on a configuration directive
3. applets that use non-standard configuration names (like `CONFIG_APP_UDHCPD` instead of `CONFIG_UDHCPD` in some versions of BusyBox)
4. features

For some applets aliases are installed by default as symlinks. These aliases are recorded in the binary, but there is no separate applet for it. In the BusyBox sources (1.15.2, others might be different) these are defined as:

```
IF_CRYPTPW(APPLET_ODDNAME(mkpasswd, cryptpw, _BB_DIR_USR_BIN,  
_BB_SUID_DROP, mkpasswd))
```

So if the `cryptw` tool is built, an additional symlink called `mkpasswd` is added during installation.

If extra functionality is added to an applet in BusyBox it is defined in the source code by macros like the following:

```
IF_SHA256SUM(APPLET_ODDNAME(sha256sum, md5_sha1_sum, _BB_DIR_USR_BIN,  
_BB_SUID_DROP, sha256sum))  
IF_SHA512SUM(APPLET_ODDNAME(sha512sum, md5_sha1_sum, _BB_DIR_USR_BIN,  
_BB_SUID_DROP, sha512sum))
```

The above configuration tells to add extra symlinks for `sha256sum` and `sha512sum` if BusyBox is configured for support for the SHA256 and SHA512 algorithms. The applet that implements this functionality is `md5_sha1_sum`.

Non-standard configuration names can be fixed by using a translation table that translates to the non-standard name. The current code has a translation table for BusyBox 1.15 and higher.

Detecting features is really hard to do in a generic way. In most cases it will even be impossible, because there are no clear markers (strings, applet names) in the binary that indicate that a certain feature is enabled. In cases there are clear marker strings these would still need to be linked to specific features. One possibility would be to parse the BusyBox sources and link strings to features, for example (from BusyBox 1.15.3, `editors/diff.c`):

```
#if ENABLE_FEATURE_DIFF_DIR
    diffdir(f1, f2);
    return exit_status;
#else
    bb_error_msg_and_die("no support for directory comparison");
#endif
```

The string "no support for directory comparison" only appears if the feature `ENABLE_FEATURE_DIFF_DIR` is not enabled.

Implementing this will be a lot of work and it will likely not be very useful.

D.6 Feeding information into the tool

By referencing with information extracted from the standard BusyBox sourcecode it is possible to get a far more accurate configuration, because it is known which applets use which configuration, unless:

- new applets were added to BusyBox
- applets use old names, but contain different code

The names of applets that are defined in BusyBox serve as a very good starting point. How these are recorded in the sources has changed a few times and depends on the version of BusyBox. The tool `appletname-extractor.py` can extract these from the BusyBox sources and store them for later reference as a simple lookup table in Python pickle format.

Names of applets per version breakdown:

- 1.15.x and later: `include/applets.h` IF syntax
- 1.1.1-1.14.x: `include/applets.h` USE syntax
- 1.00-1.1.0: `include/applets.h` (different syntax)
- 0.60.5 and earlier: `applets.h`, like 1.00-1.1.0 but with a slightly different syntax

In one particular version of BusyBox (namely 1.1.0) there is a mix of three different syntaxes: (0.60.5, 1.00 and another) for a few applets (`runlevel`, `watchdog`, `tr`).

There are also a few applets in 1.1.0 which seem to be a bit harder to detect: `busybox`, `mkfs.ext3`, `e3fsck` and `[[`. These can easily be added by hand, since there are just four of them.

Another issue that is currently unresolved is that not all the shells are correctly recognized.

D.7 Extracting configurations from BusyBox sourcecode

The `busybox.py` script makes use of a table that maps applet names to configuration directives. These tables are stored in a Python pickle and read by `busybox.py` upon startup. To generate these pickle files the `appletname-extractor.py` should be used. In the standard distribution for BAT the configurations for most versions of BusyBox are shipped.

The applet names are extracted from a file called `applets.h` or `applets.src.h`.

```
python appletname-extractor.py -a /path/to/applets.h -n $VERSION
```

The configuration will be written to a file `$VERSION-config` and should be moved into the directory containing the other configurations.

E Linux kernel scripts internals

The Linux kernel processing scripts are at the moment still very experimental. This is because there are a few challenges when working with Linux kernel source code and Linux kernel binaries. Because there are so many variants of the Linux kernel plus associated kernel drivers floating around (vendor versions, specific hardware ports, out of tree kernel drivers, etcetera) it is difficult to do a really good job.

Although the scripts are right now doing a “good enough” job for core Linux kernel functionality, it will take a few more iterations to declare it fit for production use.

E.1 Extracting visible strings from the Linux kernel binary

If a kernel is an ELF binary (sometimes) the relevant sections of the binary can be read using `readelf`. Otherwise `strings` can be run on the binary. This method will return more strings than if using `readelf`, but the extra strings are only extra cruft that won't be matched.

E.2 Extracting visible strings from a Linux kernel module

If a kernel module is an ELF binary (most cases) the relevant sections of the binary can be read using `readelf`. Otherwise `strings` can be run on the binary. This method will return more strings than if using `readelf`, but the extra strings are only extra cruft that won't be matched.

E.3 Extracting strings from the Linux kernel sources

The Linux kernel is full of strings that can end up in a binary. Some programmers have defined macros just specific to their part of the kernel for ease of use (often

a wrapper around `printk`, other programmers use more standard mechanisms like `printk`. Most strings can be extracted from the Linux kernel using `xgettext`. A minority of strings needs to be extracted using a custom regular expression.

The following two cases are worth a closer look:

E.3.1 EXPORT_SYMBOL and EXPORT_SYMBOL_GPL

The symbols defined in the `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros end up in the kernel image. The `EXPORT_SYMBOL_GPL` symbol could be interesting for licensing reporting as well, since anything that uses this symbol should be released under the GPLv2. This is a topic for future research.

E.3.2 module_param

The names of parameters for kernel modules can end up in the kernel, or in the kernel module itself. The names of these parameters are typically prefixed with the name of the module (which is often, but not always) and a dot, but without the extension of the file. In cases where the module name does not match the name of the file it was defined in extra information from the build system needs to be added to determine the right string. This is future work.

The code for this is in the function `__init param_sysfs_builtin` in `kernel/params.c`.

E.4 Forward porting and back porting

There are some strings we scan for which might not be present in certain versions, because they were removed, or not yet included in the mainline kernel. A good example is `devfs`. This subsystem was removed in Linux kernel 2.6.17, but it is not safe to assume that this was done for every 2.6.17 (or later) kernel that is out in the wild, since some vendors might have kept it and ported it to newer versions (forward porting). Similarly code from newer kernels might have been included in older versions (backporting).

E.5 Corner cases

Sometimes a `#define` or some configuration directive causes that our string matching method will not work, because the string is prepended with extra characters.

An example from `arch/arm/mach-sa1100/dma.c` from kernel 2.6.32.9:

```
#undef DEBUG
#ifdef DEBUG
#define DPRINTK( s, arg... ) printk( "dma<%p>: " s, regs , ##arg )
#else
#define DPRINTK( x... )
#endif
```

Other examples include `pr_debug`, `DBG`, `DPRINTK` and `pr_info`.

To work around this there are two ways:

1. do substring matches

2. parse the source code and record where extra code is being added as in the example above and only do substring matches in a small number of cases.

Substring matching is expensive and since it only happens in a minority of cases the second method, although not trivial to implement, would be easier. This is future work.