IRMA – An Open Source Platform for Incident Response & Malware Analysis

Guillaume Dedrie[1], Fernand Lone-Sang[1], Alexandre Quint[1]

[1] Quarkslab, 71 Avenue des Ternes, 75017 Paris
{gdedrie, flonesang, aquint}@quarkslab.com
@qb_irma, http://irma.quarkslab.com

Abstract. IRMA is an open-source platform aiming at analyzing suspicious files and facilitating the quick detection of viruses, worms, trojans, and all kinds of malware. Like several automated malware analysis platforms, IRMA provides a central place where suspicious files can be tested towards major anti-viruses engines and custom analyzers (static file analyzers, sandboxes, etc.). However, an important asset of IRMA is that you keep control over where your files go and, more importantly, who gets your data: once you install IRMA on your network, your data stays on your network.

1 Introduction

The acronym IRMA stands for "Incident Response & Malware Analysis". It is an open-source platform designed to help identify and analyze suspicious files by providing a central place where those files can be tested towards major anti-virus engines. Contrary to popular automated malware analysis platforms such as VirusTotal[1], Metascan[2], Camal[3], Malwr[4] or AVCaesar[5], IRMA attaches importance to you keeping control over where your files go and, more importantly, who gets your data. Once you install IRMA on your network, your data stays on your network, samples are not shared with the anti-malware or the security industry if not desired and the results of their analysis stay private.

Testing suspicious files against major anti-virus engines is only a first step. When an analyst detects such a file, he might want to apply custom analyses and transformations on this file, like running it in a sandbox for instance, or statically analyzing the file, which requires first unpacking it most of the time. IRMA enables you to append your custom analyzers (static file analyzers, sandboxes, etc.) and your own tools (unpackers, disassemblers, etc.) to ones that are available and shared by the community with the objective of assisting a malware analyst in extracting as relevant information as possible from a suspicious file.

Furthermore, today's defense is not only about analyzing file. IRMA can help you in getting a fine overview of the incident you dealt with: where and when a malicious file has been seen, who submitted a hash you keep a watch on, where in your information system a hash has been found, which anti-virus detects it, etc.

IRMA is still a young project. Up to now, we focused our efforts on instrumenting multiple anti-virus engines running either on Microsoft Windows or GNU/Linux systems. Thus, in this lab, we describe first the overall architecture of IRMA, which has been designed as a 3-part system. Then, we guide you in setting up your own platform inside virtual machines. Finally, we develop together a new analyzer and include it to your own IRMA setup. By the end of the lab, if you want to support this ambitious project or to reuse it, feel free to join the community: to contribute to it by submitting the analyzer you have developed or to come to see us and discuss the mechanics under the hood.

- Visit our homepage: http://irma.quarkslab.com
- Clone the sub-projects: https://github.com/quarkslab/{irma-frontend, irma-brain, irma-probe}
- Follow us on twitter: @qb_irma
- Do not hesitate to ask on IRC: #qb_irma@freenode

[1] VirusTotal - Free Online Virus, Malware and URL Scanner – https://www.virustotal.com/
[2] Metascan Online - Free file scanning with multiple antivirus engines – https://www.metascan-online.com/
[3] Camal - COSEINC Automated Malware Analysis Lab – https://camal.coseinc.com
[4] Malwr - Malware Analysis by Cuckoo Sandbox – https://malwr.com/
[5] AVCaesar – https://avcaesar.malware.lu

2 Overall Architecture

IRMA has been designed as a 3-part system: one or multiple *frontend*s where you submit suspicious files and retrieve analysis results; an analysis dispatcher referred to as the *brain*; one or multiple analyzers called *probes* registered with the brain and hosted on one or multiple machines.

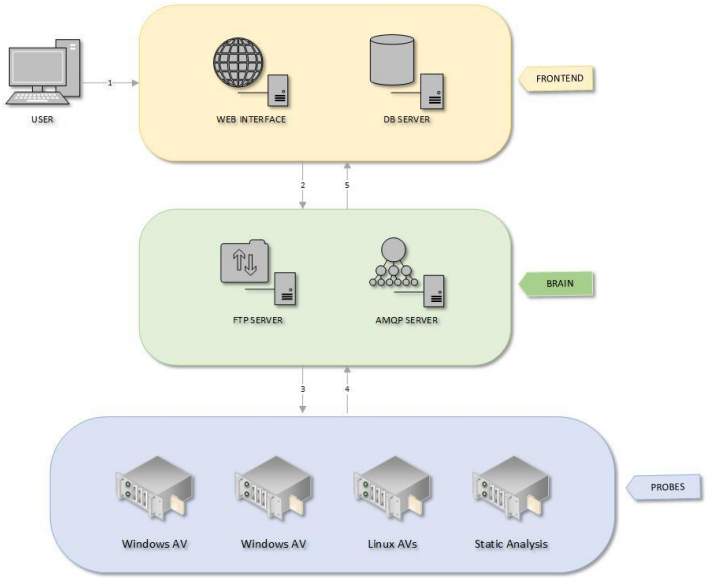


Fig. 1. File Analysis Workflow

Figure 1 describes IRMA's workflow to analyze a suspicious file. An analysis begins after an end user uploads, one or multiple files to the frontend and selects the desired analyzes to be performed. Uploads to the frontend, which exposes a *Restful API,* are performed using a *client*. When the scan is launched, the frontend checks for existing files and results first in its *SQL database*. If uploaded files are new or whether a rescan is required, it stores the uploaded files, uploads them to the FTP-TLS server on the *brain* and schedules analyzes on the *analysis dispatcher* worker using asynchronous jobs. The worker on the brain splits the analysis into analyzes subtasks that are queued for the required analyzers on the *probes*. Once processed, workers at the probe send back results to the brain, which forwards them back to the frontend. Raw results are stored in a *No-SQL database* (linked to the SQL database) and the Restful API filters these results in order to return only relevant information to the user.

2.1 The WebUI and CLI clients

The `irma-frontend` subproject comes with two clients: a web user interface client and a command line interface client. Both clients query the Restful API to upload files to be analyzed, to schedule analysis jobs and to get their results. The CLI client displays to the user the raw results returned by the API on the frontend. The WebUI is user-friendlier. It is composed of static files served by an Nginx web server. It relies heavily on AngularJS to query the API exposed by the frontend and handle dynamic views in web-applications.

One interesting feature that we have implemented on these clients and, to the best of our knowledge, does not exists in other platform, is the analysis of a group of files in once. Uploading a file one at the time and analyzing their results one by one can be tedious. You may want to scan a bunch of files linked to a malware for instance (a malware family, different payloads, etc.), to have a global view of the results and to compare them.

2.2 The Restful API

The Restful API is a python application served by an UWSGI server. It enables clients to perform queries on the SQL and the No-SQL database servers in order to retrieve analysis results or to search for relevant information. Furthermore, it can be used to schedule asynchronous analysis jobs on the brain. The latter relies on Celery, an asynchronous task queue or job queue based on distributed message passing.

Conscious that each analyst may want to combine data in different ways, we designed the API to be modular enough: one can add extra modules and extra routes to the API to answers to his analysis needs.

2.3 The Analysis Dispatcher

The analysis dispatcher is the central nerve of the IRMA platform. The intelligence of the whole platform resides in this component, which is why we compare it to a brain and called it that way. It also relies on a Celery worker to distribute analysis jobs to the available analyzers, hosted on probes. Each analyzer can register itself to the brain by creating a job queue where the brain can push orders to him.

Currently, all analyzers are at the same level. It is not possible yet to automatically dispatch to a specific probes according to the file's mime-type for instance. At the long term, we would like to allow the analysts to define their own recipes along with their analysis, with the objective to let them tune their analysis the way they want it to be.

2.4 The Registered Analyzers

As mentioned previously, analyzers can register themselves with the brain by creating their own job queues. Consequently, new analyzers can be developed with minimum efforts with the associated results auto-magically exposed by the frontend. So far, `irma-probe` subproject is bundled with 11 analyzers.

- 8 anti-viruses analyzers:

Analyzer Name	Anti-Virus Name	Description	Platform
ClamAV	ClamAV	Instruments ClamAV Daemon	GNU/Linux
ComodoCAVL	Comodo Antivirus for Linux	Instruments Comodo Free Antivirus for Linux	GNU/Linux
EsetNod32	Eset Nod32 Business Edition	Instruments Eset Nod32 Business Edition	GNU/Linux
FProt	F-Prot	Instruments F-Prot Antivirus	GNU/Linux
McAfeeVSCL	McAfee VirusScan Command Line	Instruments McAfee	GNU/Linux Microsoft Windows
Sophos	Sophos	Instruments Sophos	Microsoft Windows
Kaspersky	Kaspersky Internet Security	Instruments Kaspersky Internet Security	Microsoft Windows
Symantec	Symantec Endpoint Protection	Instruments Symantec Endpoint Protection	Microsoft Windows

- 1 file hash database for Microsoft Windows:

Analyzer Name	Database	Description
NSRL	National Software Reference Library	collection of digital signatures of known, traceable software applications

- 1 metadata analyzer:

Analyzer Name	Analyzer	Description
StaticAnalyzer	PE File Analyzer	PE File analyzer adapted from Cuckoo Sandbox

- 1 external site:

Analyzer Name	Analysis Platform	Description
VirusTotal	VirusTotal	Report is searched using the sha256 of the file which is not sent

3 Installation Procedure

3.1 Hardware requirements

IRMA can be split into a 3-part system: the frontend, the brain and the probes. Depending on how you intend to use the platform and the kind of probes you intend to use, the 3 components may or may not be installed on a unique host.

The frontend and the brain must be installed on a GNU/Linux system, preferably on a Debian distribution, which is supported and known to work. According to the kind of probes and their dependencies, each analyzer can be installed on separate hosts or share the same host as far as they do not interfere with each other[6]. So forth, only Debian distributions and Microsoft Windows 7 systems have been tested for probes.

We cannot give you any specific hardware requirements. On one hand we managed to run the whole IRMA platform on a single machine by hosting it with multiple systems inside virtual machines: this setup gives fairly high throughput as long as it has reasonable IO (ideally, SSDs), and a good amount of memory (our setup was an i7 cpu with 16 GB ram on regular drives (at least 200 GB required), on the other hand, a lighter version of the system with the three parts together and only a few probes (`Clamav`, `FProt`, `StaticAnalyzer` and `Virustotal`) was successfully installed on a single virtual machine (1 GB of Ram and 4 virtual processors).

3.2 Automating IRMA Installation with Vagrant

The installation procedure for each component is documented in their respective documentation[7]. IRMA heavily relies upon software components that must be configured specifically to serve the platform. As the configuration of these components can be tedious, time consuming and error prone, we searched for a better way to hit the goal of higher agility and faster code deployment. We have turned to DevOps, a variety of techniques, tools, and methodologies employed to make developers and operations work together in order to hit higher speeds and take advantage of larger and larger scale infrastructures. This section describes how to automate and install a whole up-to-date IRMA platform within virtual machines in a matter of minutes.

3.2.1 Prerequisites

In order to provision and deploy IRMA platform for this lab, you need to install the following software:

- Vagrant[8], version 1.5 or higher
- VirtualBox[9] Virtual Machine Manager, as it is used by default by Vagrant
- Ansible, version 1.6 or higher

The Vagrant files and the Ansible playbooks to install IRMA will be released during the conference.

---

[6] For instance, we managed to host several GNU/Linux anti-viruses on a unique probe by preventing it to launch daemons at startup. This is difficult for Microsoft systems on which it is not recommended to install multiple anti-viruses.

[7] IRMA Documentations – http://irma-frontend.rtfd.org/; http://irma-brain.rtfd.org/; http://irma-probe.rtfd.org/

[8] http://www.vagrantup.com/

[9] https://www.virtualbox.org/

3.2.2 Cloning the repositories

IRMA subprojects are hosted on github.com. Make sure to clone all the repositories with their dependencies:

```
$ mkdir IRMA
$ pushd IRMA
$ git clone --recursive https://github.com/quarkslab/irma-frontend.git
$ git clone --recursive https://github.com/quarkslab/irma-brain.git
$ git clone --recursive https://github.com/quarkslab/irma-probe.git
$ git clone --recursive https://github.com/quarkslab/irma-ansible-provisioning.git
$ popd
```

Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize the productivity and flexibility of you and your team. Machines are provisioned on top of VirtualBox and provisioning tools such as Ansible can be used to automatically install and configure software on the machine. The provided Vagrant file expects a specific directory layout. Make sure you have the following directory layout after cloning the repositories:

```
$ tree -L 1 IRMA
IRMA
├── irma-ansible-provisioning
├── irma-brain
├── irma-frontend
└── irma-probe
```

If you choose to change the directory layout, make sure to change the YAML files in the `irma-ansible-provisioning/environments/` folders accordingly. The options that must be adapted are variables prefixed with "`share_`", which indicate to vagrant where to get the application of each subsystem on your computer.

3.2.3 Preparing your virtual machines with Vagrant

The provided Vagrant file defines setups for 3 environments, described in the YAML files located in `irma-ansible-provisioning/environments/`:

- `dev`: install each component in separate virtual machines for development purpose;
- `prod`: install each component in separate virtual machines for production purpose;
- `allinone_dev`: install all components in a single virtual machine for development purpose;
- `allinone_prod`: install all components in a single virtual machine for development purpose.

Along this lab, we will be using the `allinone_dev` environment, as we will develop together custom probes. Make sure to define the `VM_ENV` environment variable for "`allinone_dev`" or change the `VM_ENV` variable in the `VagrantFile` accordingly, then run Vagrant:

```
$ export VM_ENV="allinone_dev"
$ sed -i 's/env =.*$/env = "allinone_dev"/' VagrantFile
$ vagrant up --no-provision
```

Once the master box file has been downloaded, Vagrant should have set up a virtual machine ready to be provisioned.

3.2.4 Provisioning your Virtual Machine

By default, the virtual machine is provisioned with default configuration files bundled with IRMA. As the INI configuration files located at `irma-{brain, frontend, probe}/config/*.ini` are transferred from the host to the virtual machine, you will need to locally modify them to match the user and password defined in the configuration files in the `irma-ansible-provisioning/group_vars/*` folder.

You are ready now to provision your virtual machine. The following command will download and configure all the required components IRMA relies upon:

```
$ sudo ansible-galaxy install -r galaxy.yml
$ vagrant provision
```

3.2.5 Reaching the WebUI and submitting files

The Nginx server, which serves the Web User Interface, is configured on a specific virtual host. Update your `/etc/hosts` file and add the following line:

```
$ 172.16.1.30        www.frontend.irma
```

The WebUI you have just installed should be available at the following address: http://www.frontend.irma. Let us note that the WebUI Nginx configuration can be quickly changed to support TLS. Please refer to the documentation of `irma-ansible-provisioning` for the procedure.

4 Integration of a Custom Analyzer

In IRMA we call analyzer a python module able to output data from a given source file. Turning an analyzer into a functional probe requires first to turn the analyzer module into an IRMA plugin. The benefit of using a plugin is that, on every celery daemon restart, the plugins directory is scanned and all available plugins are loaded and turned into functional probes able to receive files to analyze from the brain.

With this approach, the celery part is separated from the processing part and the analyzer could be easily used separately from IRMA.

4.1 IRMA probe source tree

A typical probe install will have all IRMA related files placed into `/opt/irma/irma-probe` directory.

```
$ tree -L 1 /opt/irma/irma-probe
/opt/irma/irma-probe
├── config                          contains configuration files to connect to the brain
├── docs                            contains all the documentation files (need sphinx to build it)
├── lib                             contains all the common python modules for frontend, brain and probe
├── MANIFEST.in
├── modules                         contains all the analyzers plugins shipped with IRMA
├── probe                           contains the celery worker core file (tasks.py)
├── requirements.txt                all python dependencies
├── setup.cfg
└── setup.py                        setup-tools script for package generation
```

4.2 Create an empty plugin - the hello world probe

First of all, create a new category under `modules` and call it `custom`. Turn this directory into a python module by creating an empty file named `__init__.py`.

Create an empty plugin directory named `helloplugin` and turn it into a python module like before.

The source tree should look like this:

```
modules
├── custom
│   ├── __init__.py
│   └── helloplugin
│       └── __init__.py
 […]
```

Now it is time to create our plugin core file. Create a new python source file called `plugin.py` under `helloplugin` directory with the following content:

```
from datetime import datetime
from lib.common.compat import timestamp
from lib.plugins import PluginBase
from lib.plugin_result import PluginResult


class HelloPlugin(PluginBase):

    # =================
    #  plugin metadata
    # =================

    _plugin_name_ = "HelloPlugin"
    _plugin_author_ = "Me"
    _plugin_version_ = "1.0.0"
    _plugin_category_ = "custom"
    _plugin_description_ = "Hello World Plugin"
    _plugin_dependencies_ = []

    # =============
    #  constructor
    # =============

    def __init__(self):
        pass

    # ==================
    #  probe interfaces
    # ==================

    def run(self, paths):
        # create a generic return dict
        ret = PluginResult(name=type(self).plugin_name,
                           type=type(self).plugin_category,
                           version=None)
        started = timestamp(datetime.utcnow())
        ret.results = "Hello World"
        stopped = timestamp(datetime.utcnow())
        ret.duration = stopped - started
        ret.status = 0
        return ret
```

As you could see the plugin part is mixed with the analyzer part, which is inlined in the run function as it consists only in this line:

```
        ret.results = "Hello World"
```

(For more complex analyzer the plugin file could be separated from the analyzer file.)

Now let's try our new plugin by restarting celery daemon and see if it is well detected. A warning message should appear announcing that our plugin has been successfully loaded:

```
$sudo service celeryd.probe restart
[…]
WARNING:probe.tasks: *** [custom] Plugin HelloPlugin successfully loaded
OK
```

`tasks.py` is the celery task core file. It is in charge of scanning the plugin directory, creating a queue per registered plugins and then to dispatch files received on all this queues to the respective analyzer.

Now we could try to scan a file on the web interface and see what is happening with our new probe. If everything went fine we should get a new result entry with our `helloworld` plugin output.

4.3 Minimal response

The output format expected for probe is json. In order to have all mandatory fields, a minimal response class named `PluginResult` is used in our `helloworld` example. The fields in questions are the following:

```
{
    'name'        : str() with the name of the probe
    'type'        : str() with the category of the probe
    'version'     : str() with the version of the probe
    'platform'    : str() with the platform on which the probe is executed
    'duration'    : duration in seconds
    'status'      : return code (< 0 is error, 0 > is context specific)
    'error'       : None if no error (state > 0) else str() with the error
    'results'     : Probe results
}
```

The `results` key could hold a dictionary with all the values that you want to send back to the frontend. The main goal of the probe is to output as many values as it could.

4.4 Dependencies

IRMA contain some mechanisms to prevent loading a plugin if a dependency is not satisfied. Here are all the dependencies checks available:

Class name	Function
`BinaryDependency`	Check for specified binary in the current path
`ModuleDependency`	Check if specified module could be imported
`FileDependency`	Check if the specified path exists and is a file
`FolderDependency`	Check if the specified path exists and is a directory
`PlatformDependency`	Check if we are running on specified platform

Let’s now turn our `helloworld` plugin into a file type guesser named `TypeGuesser`. For that purpose we will use `python-magic` module. First we move our new plugin into the `metadata` category. Create a new directory named `type_guesser` and copy our `helloworld` plugin into it.

In order to handle correctly the dependency we will add a `ModuleDependency`. We will declare it in the plugin metadata by adding the following lines:

```
    _plugin_name_ = "TypeGuesser"
    _plugin_author_ = "Me"
    _plugin_version_ = "1.0.0"
    _plugin_category_ = IrmaProbeType.metadata
    _plugin_description_ = "File type guesser based on python-magic"
    _plugin_dependencies_ = [
        ModuleDependency(
            'magic',
            help='See requirements.txt for needed dependencies'
        ), ]
```

And load the dependency only at plugin initialization:

```
    def __init__(self):
        self.module = sys.modules['magic']
```

Now we just restart the celery daemon:

```
$ sudo service celeryd.probe restart
[…]
WARNING:root: *** [plugin] Plugin failed to load: TypeGuesser miss dependencies: magic (Mo-
duleDependency). See requirements.txt for needed dependencies
[…]
```

As the dependency is not satisfied, the plugin is not loaded and the help message from the dependency handler is printed. Let's create a `requirements.txt` file to easily install our new plugin. Fill it with the following content:

```
python-magic>=0.4.6
```

And install it.

```
$ sudo pip install -r modules/metadata/type_guesser/requirements.txt
```

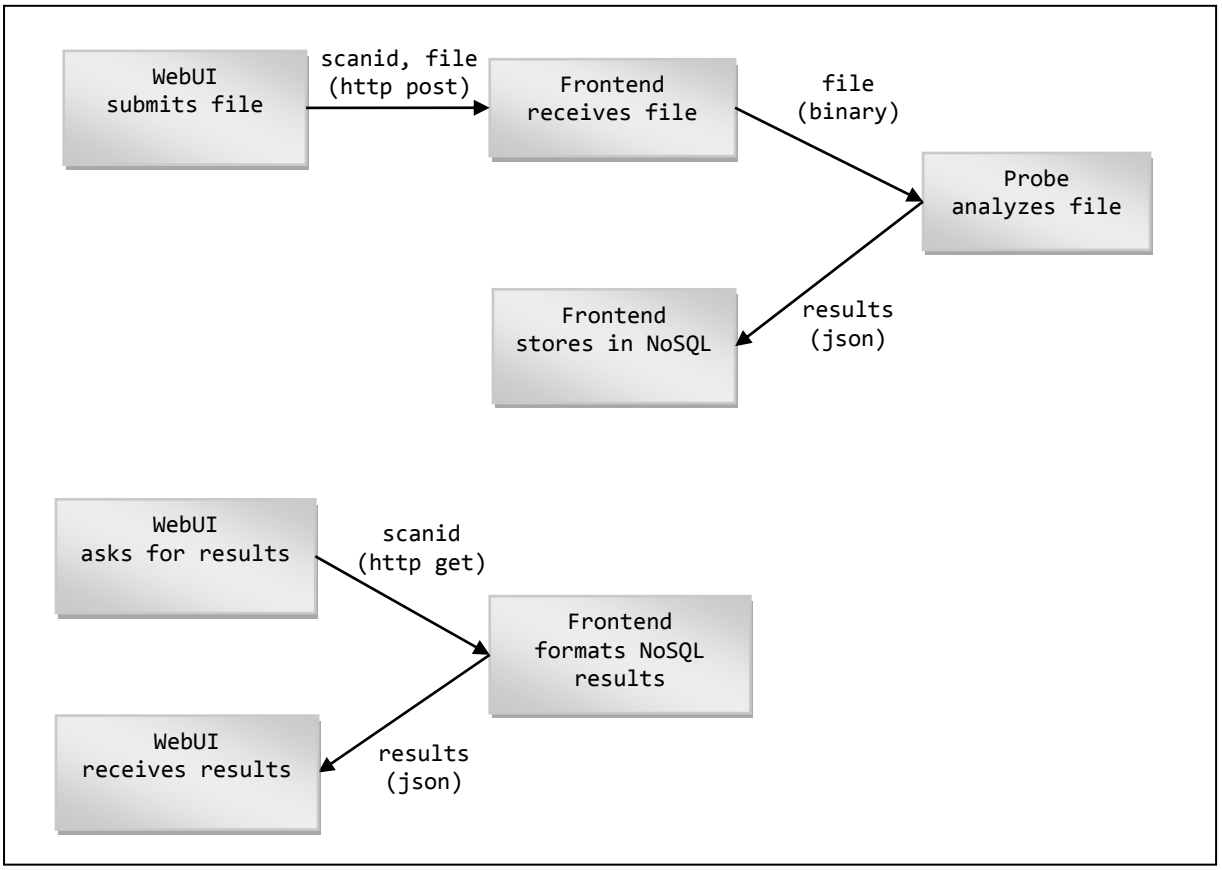
Now if we restart the probes it should works:

```
$ sudo service celeryd.probe restart
[…]
WARNING:probe.tasks: *** [metadata] Plugin TypeGuesser successfully loaded
[…]
```

4.5 Frontend Part

4.5.1 Results workflow

Here is an incomplete overview of the file processing workflow. The important thing to notice is that the probe json results are stored as it is in the No-SQL database, but could be filtered before being displayed on any interfaces through the formatters. In this section we will see how to use them.



4.5.2 Formatter introduction

In frontend source directory you will find a similar plugin tree as the probe `modules` directory tree but under `frontend/helpers/formatters` directory. It should look like:

```
frontend
├── helpers
│   └── formatters
│       ├── antivirus
│       └── external
│           └── virustotal
[…]
```

Every probe that needs special formatting could have a formatter. Each formatter plugin will declare what plugin it can handle. For example the default antivirus formatter handles all probes with type `IrmaProbeType.antivirus`:

```
    @staticmethod
    def can_handle_results(raw_result):
        return raw_result.get('type', None) == IrmaProbeType.antivirus
```

You can also have a formatter that just handles the result of one specific probe. For example `VirusTotal` formatter handle only results from the `VirusTotal` probe:

```
    @staticmethod
    def can_handle_results(raw_result):
        expected_name = VirusTotalFormatterPlugin.plugin_name
        expected_category = VirusTotalFormatterPlugin.plugin_category
        return raw_result.get('type', None) == expected_category and \
            raw_result.get('name', None) == expected_name
```

The main method of formatter plugin is the format method that transforms the probe raw results into filtered results (that still should have all json keys seen in chapter 4.3):

```
    @staticmethod
    def format(raw_result):
       […]
       return raw_result
```

You could change all values but mandatory keys should still be present in the returned result dict.

4.5.3 Formatter example

For example the `VirusTotal` formatter plugin will take the raw results from `VirusTotal` probe and only output the ratio of detection.

```
    @staticmethod
    def format(raw_result):
        status = raw_result.get('status', -1)
        vt_result = raw_result.get('results', {})
        if status != -1:
            av_result = vt_result.get('results', {})
        if status == 1:
            # get ratios from virustotal results
            nb_detect = av_result.get('positives', 0)
            nb_total = av_result.get('total', 0)
            raw_result['results'] = "detected by {0}/{1}" \
                                    "".format(nb_detect, nb_total)
```

5 Conclusion and Future Work

There are a lot of new features to come. So far we focused our efforts on providing a stable architecture that could easily be installed and customized.

On the frontend part, we have planned to add more views and search methods in order to get more information from files already present in database.

On the analysis part (brain), we wanted to add support for more antivirus engines and sandbox probe. The next big feature to come is a probe dispatcher that will be in charge of chaining probe analyzes. For example, there is no point in sending a non `PE` file to the static analysis, but it could be valuable to resend a packed binary to the antivirus engines after unpacking.

If you intend to create your own probe, you can send it to us and we will happily integrate it in the default plugin list shipped with IRMA. If you also want to contribute to the whole project you are welcomed.

Acknowledgments

This project is co-funded by the following actors: CEA DAM, DCNS, GOVCERT.LU (governmental CERT of Luxembourg), Airbus Group, QuarksLab and Orange Group IS&T. We would like to thank all the contributors, people who have submitted patches, reported bugs, helped answer newbie questions, and generally made IRMA that much better.