`

RED PANDA USER DOCUMENTS

Andrew Meng, Jake Milanowski, Simon Snider, Danielle Villa

TABLE OF CONTENTS

INTRODUCTION

This document provides instructions on how to install, run, and develop with Red Panda. The Installation Guide gives instruction on how to install Red Panda using both Dockerfile and manual methods. The User Guide details how to run Red Panda and view its outputs. The Development & Maintenance section describes how to develop Red Panda, future features that could be implemented, as well as a troubleshooting guide that provides advice should problems arise during development. The Deployment Strategy includes a checklist of questions to be asked when deploying a version of Red Panda as well as the plan implemented during its first release. The configuration management plan describes the process used to organize and manage the source control repository for Red Panda.

INSTALLATION GUIDE

Introduction

There are two ways to install Red Panda. The system can be run from a Dockerfile or the system and its dependencies can be manually installed. This section describes both methods and how to use them.

Dockerfile method

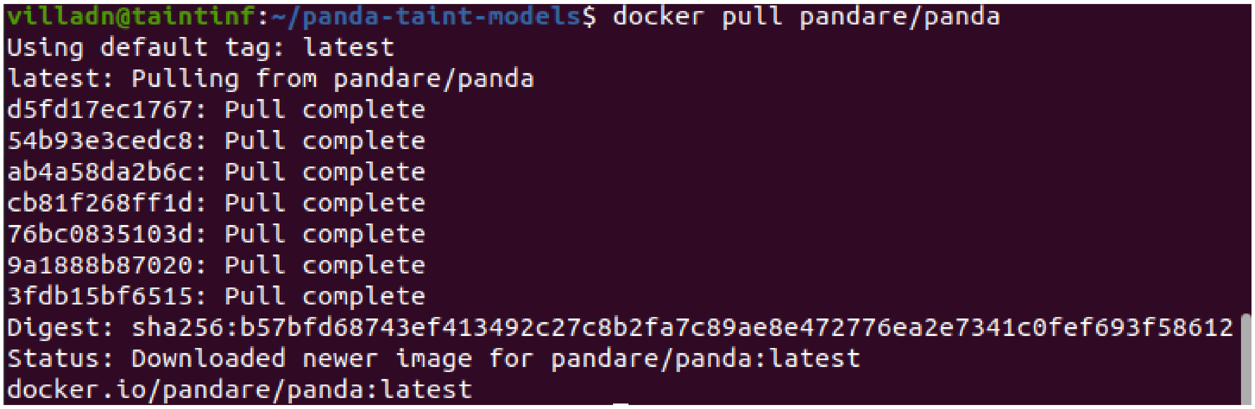
External dependencies required by this installation method:

- Ubuntu 20.04 or above (installation description at https://ubuntu.com/)
- Docker (installation description at https://docs.docker.com/engine/install/ubuntu/)

Additionally, the latest docker image for both pandare and python. Unfortunately, it is not possible to choose between versions of PANDA when running through docker. Once docker has been installed, use the following commands to get them:

```
docker pull pandare/panda
docker pull python
```

A successful execution of the pull is shown in figure 1:



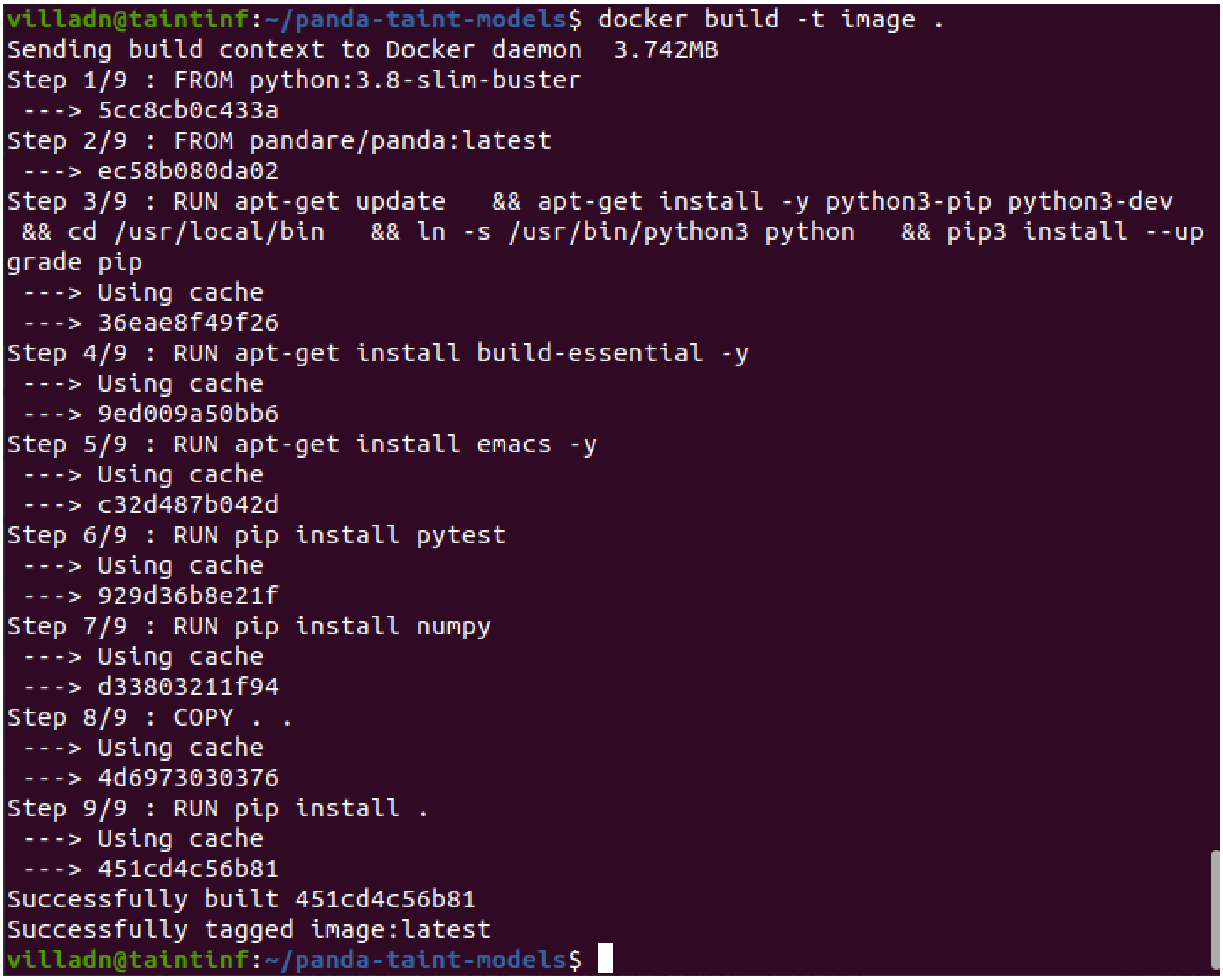
```
villadn@taintinf:~/panda-taint-models$ docker pull pandare/panda
Using default tag: latest
latest: Pulling from pandare/panda
d5fd17ec1767: Pull complete
54b93e3cedc8: Pull complete
ab4a58da2b6c: Pull complete
cb81f268ff1d: Pull complete
76bc0835103d: Pull complete
9a1888b87020: Pull complete
3fdb15bf6515: Pull complete
Digest: sha256:b57bfd68743ef413492c27c8b2fa7c89ae8e472776ea2e7341c0fef693f58612
Status: Downloaded newer image for pandare/panda:latest
docker.io/pandare/panda:latest
```

Figure 1: An example of a successful pull of panda's docker image

Next, clone the Red Panda repository from https://github.com/SimonSnider/RedPanda. In the repository, there is a Dockerfile that uses these images to build the dependencies needed for Red Panda. Building and running a shell in the Dockerfile may require root privileges but will allow you to run Red Panda:

```
docker build -t image .
```

A successful docker build may take upwards of 5 minutes if it has not been built before. The process is occasionally updated in the command line as in figure 2:



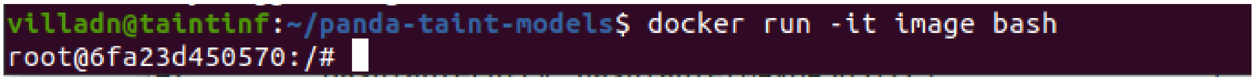
```
villadn@taintinf:~/panda-taint-models$ docker build -t image .
Sending build context to Docker daemon  3.742MB
Step 1/9 : FROM python:3.8-slim-buster
 ---> 5cc8cb0c433a
Step 2/9 : FROM pandare/panda:latest
 ---> ec58b080da02
Step 3/9 : RUN apt-get update   && apt-get install -y python3-pip python3-dev
 && cd /usr/local/bin   && ln -s /usr/bin/python3 python   && pip3 install --up
grade pip
 ---> Using cache
 ---> 36eae8f49f26
Step 4/9 : RUN apt-get install build-essential -y
 ---> Using cache
 ---> 9ed009a50bb6
Step 5/9 : RUN apt-get install emacs -y
 ---> Using cache
 ---> c32d487b042d
Step 6/9 : RUN pip install pytest
 ---> Using cache
 ---> 929d36b8e21f
Step 7/9 : RUN pip install numpy
 ---> Using cache
 ---> d33803211f94
Step 8/9 : COPY . .
 ---> Using cache
 ---> 4d6973030376
Step 9/9 : RUN pip install .
 ---> Using cache
 ---> 451cd4c56b81
Successfully built 451cd4c56b81
Successfully tagged image:latest
villadn@taintinf:~/panda-taint-models$ 
```

Figure 2: An example of successfully building Red Panda's docker image

After the image is built, it can be run using the following command:

```
docker run -it image bash
```

Once the run begins you should see something similar to Figure 3:



```
villadn@taintinf:~/panda-taint-models$ docker run -it image bash
root@6fa23d450570:/#
```

Figure 3: an example of what the terminal should look like upon running the docker image

Then you should be able to successfully run:

```
python3 red_panda/REDPANDA.py <arguments>
```

See the User Guide for more details about the potential arguments to the system.

Developer method (advanced)

This method is not recommended as it does not guarantee a stable release.

Dependencies:

- Ubuntu 20.04 or above (installation description at https://ubuntu.com/)
- Python 3.7 or above (installation description at https://www.python.org/)

In the instances that the docker file format does not perform a necessary feature for your development purposes, a manual installation can help by providing more comprehensive system modification.

To get started, install and build your desired panda release from scratch. There are many ways to install Panda, but the most comprehensive resource can be found at: https://github.com/panda-re/panda

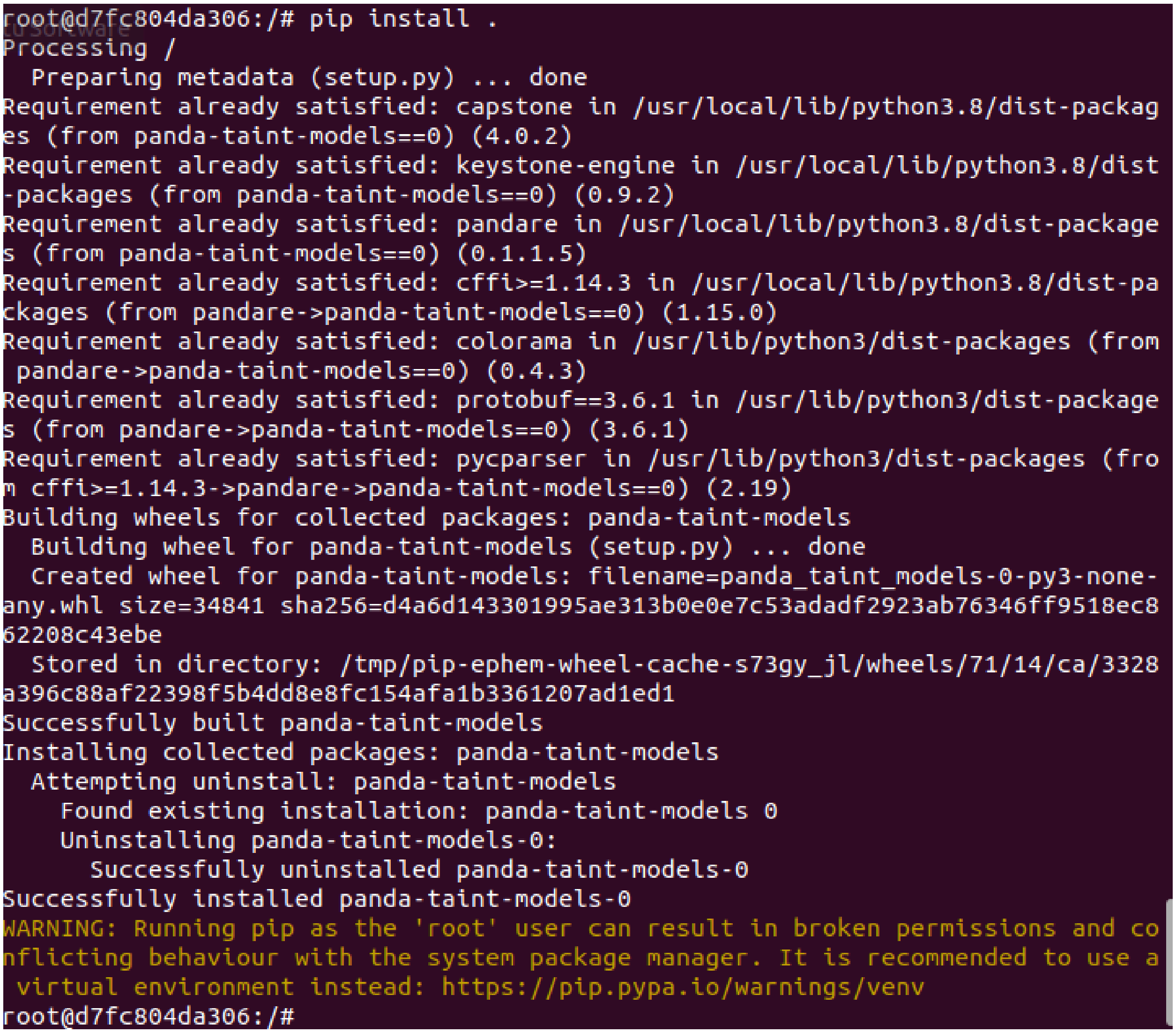
Make sure that the corresponding python pandare packages are installed, which can be checked using the following command:

```
pip install pandare
```

Once this is complete, clone the Red Panda repository. Navigate to the parent folder of the project. Then run:

```
pip install .
```

An example of a successful pip install is shown in figure 4:



```
root@d7fc804da306:/# pip install .
Processing /
  Preparing metadata (setup.py) ... done
Requirement already satisfied: capstone in /usr/local/lib/python3.8/dist-packag
es (from panda-taint-models==0) (4.0.2)
Requirement already satisfied: keystone-engine in /usr/local/lib/python3.8/dist
-packages (from panda-taint-models==0) (0.9.2)
Requirement already satisfied: pandare in /usr/local/lib/python3.8/dist-package
s (from panda-taint-models==0) (0.1.1.5)
Requirement already satisfied: cffi>=1.14.3 in /usr/local/lib/python3.8/dist-pa
ckages (from pandare->panda-taint-models==0) (1.15.0)
Requirement already satisfied: colorama in /usr/lib/python3/dist-packages (from
 pandare->panda-taint-models==0) (0.4.3)
Requirement already satisfied: protobuf==3.6.1 in /usr/lib/python3/dist-package
s (from pandare->panda-taint-models==0) (3.6.1)
Requirement already satisfied: pycparser in /usr/lib/python3/dist-packages (fro
m cffi>=1.14.3->pandare->panda-taint-models==0) (2.19)
Building wheels for collected packages: panda-taint-models
  Building wheel for panda-taint-models (setup.py) ... done
  Created wheel for panda-taint-models: filename=panda_taint_models-0-py3-none-
any.whl size=34841 sha256=d4a6d143301995ae313b0e0e7c53adadf2923ab76346ff9518ec8
62208c43ebe
  Stored in directory: /tmp/pip-ephem-wheel-cache-s73gy_jl/wheels/71/14/ca/3328
a396c88af22398f5b4dd8e8fc154afa1b3361207ad1ed1
Successfully built panda-taint-models
Installing collected packages: panda-taint-models
  Attempting uninstall: panda-taint-models
    Found existing installation: panda-taint-models 0
    Uninstalling panda-taint-models-0:
      Successfully uninstalled panda-taint-models-0
Successfully installed panda-taint-models-0
WARNING: Running pip as the 'root' user can result in broken permissions and co
nflicting behaviour with the system package manager. It is recommended to use a
 virtual environment instead: https://pip.pypa.io/warnings/venv
root@d7fc804da306:/#
```

Figure 4: a successful pip install

Development Environment Set Up

Development of the RED PANDA system can be done in nearly any python-compatible text editor or IDE. Our internal testing is done using emacs and vi in normal linux environments and Visual Studio Code (available at https://code.visualstudio.com/) in Windows Subsystem Linux (whose installation is described at https://docs.microsoft.com/en-us/windows/wsl/install). To get started in any of these environments follow the above instructions to install the panda system to a native or emulated Ubuntu environment. For emacs and vi, simply launch them from the command line to edit and save a file. To use VSCode, open the program from the command line by using the `code` command. Once VSCode is open, simply open the repository directory using the File > Open Folder menu, then open and edit files as usual.

Once a file has been edited it is time to merge it into the current build. Edits to the main REDPANDA.py executable or any test files require no additional steps and automatically update upon the next system run. Edits to any file in the red_panda directory require a new install. Simply open a command terminal and run `pip install .` in the repository folder to install your changes.

USER GUIDE

Introduction

This section includes instructions on how to run Red Panda and each of the settings that can be configured

RED PANDA is a system designed to determine the correctness of the taint models generated by Panda RE’s taint2 plugin. The system utilizes the pandare python package to hook into the PANDA reverse engineering suite and test instructions.

The system is run using the following command:

```
python3 red_panda/REDPANDA.py <arguments>
```

The following <arguments> must be specified on the command line, while all others are optional or use-case dependent. All arguments are specified in the help specifications or in further detail in the dedicated argument description section.

Running the command with the help argument gives a description for each of the arguments for Red Panda.

```
python3 red_panda/REDPANDA.py --help
usage: REDPANDA.py [-h] [-architecture {mips32}]
                   [-random_instructions RANDOM_INSTRUCTIONS | -bytes_file
BYTES_FILE | -instructions_file INSTRUCTIONS_FILE]
                   [-iterations ITERATIONS] [-analysis_model {0,1}]
                   [-output_model {0,1}] [-name NAME] [-v] [-i]
                   [-threshold THRESHOLD] [-seed SEED]

optional arguments:
  -h, --help            show this help message and exit
  -architecture {mips32}
                        the instruction set architecture to generate and run
                        instructions in
  -random_instructions RANDOM_INSTRUCTIONS
                        a number of random instructions to generate
  -bytes_file BYTES_FILE
                        path to a file with a list of byte-assembled
                        instructions
  -instructions_file INSTRUCTIONS_FILE
                        path to a file with a list of unassembled
instructions
  -iterations ITERATIONS
                        the number of times an instruction is randomly run
for
                        each register in the specified ISA
  -analysis_model {0,1}
                        the mathematical model used for analysis: 0 - reg-
                        correlations, 1 - mem-reg-correlations
  -output_model {0,1}   choose the model for output: 0 - matrix, 1 -
threshold
  -name NAME            the name for the current run of the system, will
                        become the name of the output file
  -v, --verbose         enable verbose mode for each system step
  -i, --intermediate    print a file containing the register contents of each
                        run
  -threshold THRESHOLD  the correlation threshold for instructions to
register
                        as tainted
  -seed SEED            the seed to generate instructions with
```

Config Mode

Sometimes it is useful to leverage the same settings over the course of multiple runs of Red Panda without needing to input the same settings on the command line each time. When this functionality is desired, you can use the Red Panda configurable mode to accomplish it. Said mode is initialized as follows:

```
python3 red_panda/REDPANDA.py @config_file.cfg
```

To use config mode simply create a new config file and enter the desired arguments for a system run. Further details on how this can be accomplished are found in the config mode example in the examples section.

System Arguments

There are a variety of arguments to pass into REDPANDA.py. Below is a full list of what they mean and the types are input they expect. For argument fields that specify a list of options, specify the list number as input during runtime.

Name (-name)

Specifies the name of the system run. Used to generate output files.

Valid Options

- Any valid string

Usage Example: `−name=default`

Architecture (-architecture)

Specifies the instructions set architecture RED PANDA is to run in. This also determines the ISA instructions are generated during run time.

Valid Options

- mips32 - The MIPS instruction set
- x86-64 - The x86-64 or AMD64 instruction set

Usage Example: `–architecture=mips32`

Instruction Source (-random_instructions, -bytes_file, -instructions_file)

Determines the source of instructions during the RED PANDA run. Instructions have three different supported sources. The -random_instructions option uses RED PANDA's internal instruction generator to randomly generate valid instructions in the system. The -bytes_file option allows you to specify a new line delimited file of assembled instructions in byte format. The -instructions_file option does the same using unassembled instructions in a chosen ISA.

Random Instructions

Valid Options

- Any positive 32-bit integer

Usage Example: `–random_instructions=186`

Bytes File

Valid Options

- the path to a file holding instruction bytes

Usage Example: `–bytes_file=bytes.txt`

Instructions File

Valid Options

- the path to a file holding unassembled instructions

Usage Example: `–instructions_file=instructions.txt`

Instruction Iterations (-iterations)

Specifies the number of times an instruction is run to collect correlation data for a particular register. Note that this means that an architecture with 32 registers will run 32 times the entered number.

Valid Options

- Any positive 32-bit integer

Usage Example: `–iterations=12`

Analysis Model (-analysis_model)

Specifies which type of analysis the system will perform. Currently reg-correlational and mem-reg-correlational perform the same analysis without or with memory tracking.

Valid Options

- 0 - generate correlations between registers only (currently only supported for basic MIPS functionality)
- 1 - generate correlations between registers and memory

Usage Example: `–analysis_model=1`

Output Format (-output_model)

This argument specifies the output format of the system. Output files are saved to the present working directory unless specified otherwise in the system arguments. The output file may be either a CSV file, which contains every correlation that Red Panda calculated from the intermediate data, or a text file, which only lists those correlations that are deemed significant by either the system or the user.

Valid Options

- 0 - matrix output (generates a matrix where ones in the matrix represent correlations between inputs in the row and outputs in the column)
- 1 - threshold output (generates a human readable message of unexpected correlations based on a p-value threshold)

Usage Example: `–output_model=1`

Verbose? (-v/--verbose)

Specifies if you wish the system to output debug and progress messages. NOTE: Verbose mode will print many messages to the console as the system runs. These messages are not saved to a file.

Usage Example: `–v`

Intermediate Output (-i/--intermediate)

The intermediate data file contains all of the raw data generated by running an instruction multiple times in JSON format.

Usage Example: `–i`

Output Threshold (-threshold)

The threshold required for threshold output to recognize and output a correlation.

Valid Options

- Any decimal value between 0 and 1

Usage Example: `–threshold=0.55`

Random Generation Seed (-seed)

A seed used by the random instruction generator. Used to ensure randomly generated instruction lists are consistent between runs.

Valid Options

- Any 32-bit integer

Usage Example: `-seed=1352389`

Examples

Creating a Configuration File from Scratch

Argument configuration files can be a useful tool for storing complex sets of system arguments. Red Panda provides functionality to store whole configurations and partial configurations in files. Below is an example of creating one of these files from scratch and using it to execute a Red Panda instance.

Begin by creating a new text file to store the custom configuration. Then open the file in a text editor.

```
touch my_config.txt
```

Once open, specify the arguments desired for the configuration. This configuration is going to use the MIPS architecture with a focus on high instruction iteration counts. To do so we construct each line to be a single argument defined using the same syntax as on the command line. The value given for the argument is found on the following line.

```
-architecture
mips32
-iterations
100
-analysis_model
1
-output_model
1
-threshold
.5
```

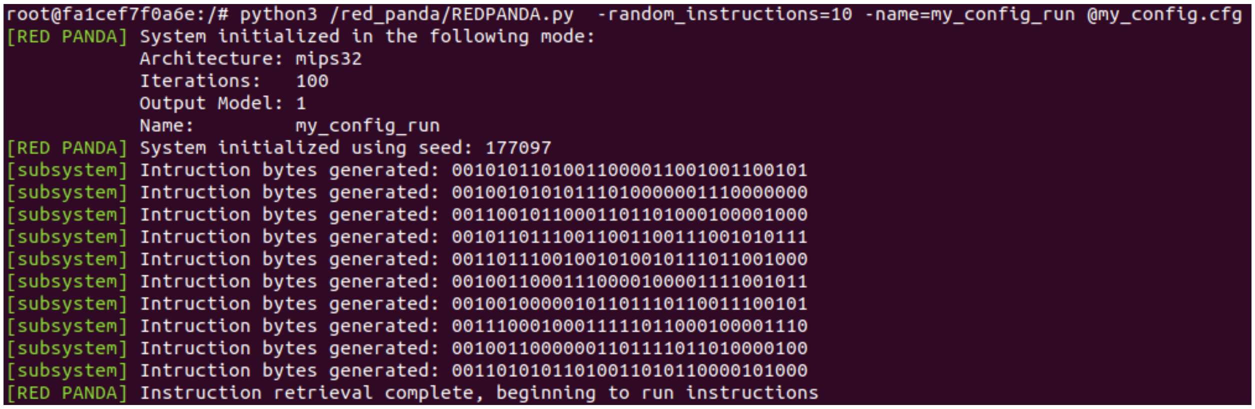
Notice that this argument list does not contain all the required arguments for a system run. Arguments not specified in the configuration file are specified during program execution, allowing for greater flexibility in configuration file uses.

Now that the file is complete, we can execute it using the following command:

```
python3 /red_panda/REDPANDA.py -random_instructions=10
-name=my_config_run @my_config.txt
```

Notice that the two required arguments of instruction source and execution name are still specified. If we wanted to keep either of these consistent between runs we could move them to the configurable file much like the others.

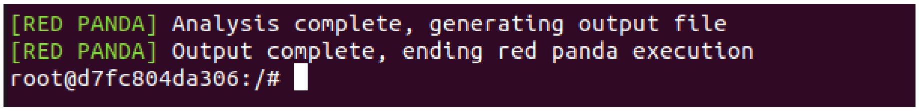
After you run the system, you should see output similar to figure 5:



```
root@fa1cef7f0a6e:/# python3 /red_panda/REDPANDA.py  -random_instructions=10 -name=my_config_run @my_config.cfg
[RED PANDA] System initialized in the following mode:
            Architecture: mips32
            Iterations:   100
            Output Model: 1
            Name:         my_config_run
[RED PANDA] System initialized using seed: 177097
[subsystem] Intruction bytes generated: 00101011010011000011001001100101
[subsystem] Intruction bytes generated: 00100101010111010000001110000000
[subsystem] Intruction bytes generated: 00110010110001101101000100001000
[subsystem] Intruction bytes generated: 00101101110011001100111001010111
[subsystem] Intruction bytes generated: 00110111001001010010111011001000
[subsystem] Intruction bytes generated: 00100110001110000100001111001011
[subsystem] Intruction bytes generated: 00100100000101101110110011100101
[subsystem] Intruction bytes generated: 00111000100011111011000100001110
[subsystem] Intruction bytes generated: 00100110000001101111011010000100
[subsystem] Intruction bytes generated: 00110101011010011010110000101000
[RED PANDA] Instruction retrieval complete, beginning to run instructions
```

Figure 5: Example output of generating random instructions

Much more output will be generated detailing the intricacies of your run. You will know the run is complete when an output message similar to figure 6 is shown:



```
[RED PANDA] Analysis complete, generating output file
[RED PANDA] Output complete, ending red panda execution
root@d7fc804da306:/#
```

Figure 6: the finish message of the red panda script

Using Non-random Instructions

Using instructions from a source other than random can be useful for getting models for a subset of an assembly language. Red Panda supports this using non-random instruction lists. In order to do so first begin by creating a file for the desired instructions to be stored.

```
touch my_instructions.txt
```

Open the file and enter the instructions which you desire to run. The entered instructions must all be in the same instruction set architecture and be viable assembly for the keystone assembler. Here the add and sub instructions are entered for the MIPS ISA. Save and close the file afterwards.

```
add $t0, $t1, $t2
sub $t4, $t2, $t6
```

Once the instructions are written it is time to run Red Panda. In order to run the system using the instruction list you must use the -instructions_file argument in place of the -random_instructions argument. The usage of which can be found as follows:

```
  Random Instructions: python3 /red_panda/REDPANDA.py
  -random_instructions=10 -name=my_instructions_run ...
```

Becomes…

```
  python3 /red_panda/REDPANDA.py
  -instructions_file=my_instructions.txt -name=my_instructions_run ...
```

DEVELOPMENT & MAINTENANCE GUIDE

Introduction

This section details the environment that Red Panda was developed in, as well as planned future changes and troubleshooting advice for the development process.

Development Environment

Development takes place in the same environment as a user's environment for testing purposes, and no specific IDE is recommended. There are no path or environments variables that need to be set; instead, environment setup is specified by the setup.py file and is done by running the `pip install .` command. However, due to python's enforcement of syntactic whitespace, any editor must use tabs instead of spaces, and each tab must be 4 spaces wide.

In order to track changes made to the system, we use git as an automated repository and UnitTest as the testing framework for regression testing. Some of the IDEs used in development include Visual Studio Code and PyCharm. For a description of the VS Code's installation process, visit https://code.visualstudio.com/docs/setup/windows.

Planned Changes

The following are features that are not implemented in the current version of Red Panda, but that may be included in future versions.

Name	**Feature Description**
ARM32	Adding additional ISAs, such as ARM32, as valid ISAs for Red Panda to analyze instructions from.
Correlation JSON	Allowing the user to specify exporting the full set of calculated coefficients as a JSON file, rather than a CSV file.
Multi-register randomizing	Randomizing more than one register at a time during the data collection stage in order to see if registers are conditionally correlated with other registers.
Sandshifter Integration	Randomly generating x86 instructions currently takes an unacceptable amount of time or results in failure. This is because x86's instruction set is sparse in the $[0\|1]^{64}$ encoding space it uses. Sandshifter uses some helpful searching heuristics to randomly generate valid instructions instead, which would result in much faster and correct generation.

Developer & Maintainer Role

The Red Panda repository is open-source, and anyone may submit a pull request to improve the codebase. However, they are asked to comply with our testing standards and all pull requests must be reviewed and approved by at least 2 active maintainers. Maintainers must be familiar with the Red Panda codebase and have experience working with it and PANDA. They will primarily be the MIT Lincoln Labs PANDA development team.

Module Details

Modules and Submodules

The following is a detailed description of each module or submodule in Red Panda. These descriptions can also be found in the Architecture and System Design section of the System Documents.

compare_to_taint

Takes the empirically-generated model and the Taint2 model as inputs, and outputs the differences between the two that are not due to our model tracking additional addresses

The outward facing function of the module is compare(). The function takes the inputs pandaModel and ourCorr. ourCorr is a Correlations object, and pandaModel is a list consisting of a 2D list describing the register to register and read to register data, and a dictionary describing the register to write data.

create_output

Takes the empirically-generated model and the comparison model as inputs, and generates output files based on user arguments

The outward facing functions of all files in the module are called generateOutput. The functions take the inputs instructionNames, data, filename, and registerNames. instructionNames is a list of instructions in plain text. Data is a Correlation object, filename is a string, and registerNames is a list of register names.

generate_instruction

Takes the user argument for the ISA and, if the user specified that a number of instructions is to be randomly generated, generates random bit strings, verifies that they are valid instructions using a disassembler, and runs them through the filter to produce a final list of random instructions

The outward facing function of the module is generateInstruction(). The function takes the inputs of instructionGenerator, filterer, and verbose. InstructionGenerator is an instruction generator data class created by the method initialize(). Filterer is an imported python filter module. Verbose is a Boolean which activates verbose mode. The output of generateInstruction() is a byte string corresponding to a generated instruction encoding.

filter

Takes in an instruction candidate and, based on its ISA, determines if it can be tainted and run by Red Panda

The outward facing function of the module is filterInstruction(). The function takes the inputs of instruction and verbose. Instruction is a string of bytes corresponding to an instruction. Verbose is a Boolean, where true is verbose mode. The output of filterInstruction() is a boolean, where true corresponds to the instruction supplied functioning for the current build of the system.

get_correlations

Takes the empirical data from the run_instruction module as input, and outputs the empirical taint model for each instruction by calculating the likelihood of each register or memory right being affected by a register or memory read.

The outward facing function of the module is the initialize() function, which takes in the RegisterStateList object produced by run_instruction module, and integer representing the number of iterations ran, a threshold for determining significant taint, verbosity, and a deprecated p-value parameter. Correlations are calulcated by running the computeCorrelations() function, which takes no arguments and outputs a Correlation object representing all taint correlations found.

models

Contains the data classes that are used by multiple modules. Any data classes used by new functionality should also to be put here.

run_instruction

Takes the instruction(s) to be run and other user arguments as inputs and orchestrates setting up PANDA, running the instructions to gather information for the empirically generated model as well as the instructions for gathering the Taint2 model and producing those models as output.

The outward facing function for the module is generateInstructionData(). The function takes the inputs of arch, instructionList, instructionIterations, and verbose. Arch is a string corresponding to the desired architecture for the system to run in. InstructionList is a list of byte string defined instructions to be run. InstructionIterations is an integer which specifies how many times an instruction is to be randomized when collecting data. Verbose is a Boolean which activates verbose mode. The output of generateInstructionData() is a list of instructionData, pandaModels, and reg_names. Instruction data contains the list of instructions run, the before states of each execution, the after state of each execution, and which registers were randomized before each execution. The pandaModels is a list where the first element is a 2D list representing the register to register and read to register correlations, and the second element is a dictionary representing the register to write correlations. The reg_names list contains the human readable names for the registers in the specified architecture, used to make output readable.

intermediary

A single script called REDPANDA.py that takes in user arguments and runs the other modules. Inputs to the system are specified using Python ArgParse and can be retrieved using the help argument. Outputs are specified using the create_output module.

Troubleshooting

Audience	**Symptom**	**Possible Cause**	**Possible solution(s)**
Developer	Modifications not showing up	Python modules were not reinstalled after modification	Run “pip install .” in the terminal in the directory containing setup.py (root)
End User	Running “Addi” in MIPS results in nonsensical correlation output	“Addi” no longer exists as of MIPS revision 6	Do not run addi.

End User	Command line arguments won't parse	Many available arguments have default values and therefore do not need to be specified, but several do not. The user must always provide an instruction file or a byte file (using instruction_file or bytes_file), a name (using name), and an architecture.	Specify these on the command line or in a configuration file referenced by the command line.
Developer	Program crashes with the error: Inconsistent use of tabs and spaces	In the line referenced, spaces are used instead of tabs, or tabs of the wrong length are used.	Remove the problematic whitespace. Set the editor's default to tabs with 4 spaces.
End User	System crashes during the execution of an instruction.	The instruction may be one that is unsupported by the system, or the system may have run out of memory.	Run the system using only that instruction. If the system still crashes, try running a different instruction. This will clarify if the issue is in the instruction, in which case it is unsupported, or the system, in which case additional debugging is required.
End User	System crashes with the error: Unrecognized arguments	No @ sign before the name of the configuration file.	Ensure that there is an @ sign before the name of the configuration file.

DEPLOYMENT STRATEGY

Introduction

This section includes a checklist of questions to be asked when deploying a version of Red Panda, as well as the plan implemented for the initial deployment of Red Panda to MIT Lincoln Labs.

Deployment Readiness Assessment Checklist

Criteria	Yes/No/N/A
Do test planning documents that describe the overall planning efforts and test approach exist?	Yes
Is testing, as specified in the test planning documents, complete?	Yes
Are test results documented?	Yes
Is the product defect-free?	Yes
Have all remaining defects been documented?	N/A
Is the product acceptance sign-off complete?	Yes
Are documents to be produced for the purpose of aiding in installation, support, or use of the product complete, published, and distributed, or are they on schedule to be completed, published, and distributed prior to deployment?	Yes
Are activities for notifying stakeholders (clients/administration) of the release on schedule to be completed as planned?	Yes
Are all new modules relevant to deployment selectable from the executable file?	Yes
Are all code segments used for exclusively debug purposes designated as being run in debug mode only?	Yes
Is the help command updated to include the most recent feature additions relevant to deployment?	Yes
Are all non-production files removed from the installation directory?	Yes
Signature: Simon Snider Date: 5/13/2022	

Deployment Plan

The nature of the system we are developing is different from creating a product for a client. Our client wants a proof of concept system that they can continue to develop after the conclusion of this project. The system consists of a package of tools downloaded and used on a command-line terminal. The only deployment it requires is creating a release branch on the GitHub and the client will create a snapshot of the repository. The true deployment of the system will be carried out by the PANDA team at some point in the future. Before turning the project over to the PANDA development team, we will include a GNU general public license (GPL). The following is the step by step deployment plan:

1. Verify all client-facing documentation including but not limited to the testing documentation, development guide, user guide, and installation guide has been coordinated for delivery to the client
2. Create an updated release branch with all final functionality of the system
3. Run a full test suite on the release branch and update relevant documents (using GitHub Actions)
4. Send the client copies of the relevant documentation
5. Verify the clients are able to create a snapshot of the repository

CONFIGURATION MANAGEMENT PLAN

Introduction

This section provides a guide to the source control repository used for Red Panda, including naming convention for branches, what types of changes constitute what kind of branch, and how to manage branches.

We will follow a git flow model through the use of a public git repository. This process mirrors the process followed by our client. Our approach will include a single development branch for features that are not yet complete, multiple feature branches for development by individuals, and additional branches for bug fixes and collaboration. Regardless of the kind of branch, most will be compartmentalized such that they exist for at most a week. We will maintain consistent naming standards for these branches in order to maintain clarity. Names will be of the form: username_mm/dd/yyyy_featureName. In this name, usernames will be lowercase and feature names will be shortened and written in camel case. Examples of branches and their purpose are shown in the following table:

Branch Name	Developer	Purpose
snidersa_02/07/2022_ImplementingX86	Simon Snider	Expanding the system to include x86
villadn_2/8/2022_dockerfile	Danielle Villa	Creating a Dockerfile for the project and its ReadMe

Once all of the features for a release have been satisfied and tested and all features have been incorporated into the main branch, a release branch will be created off of the main branch. Although no additional features will be added to preexisting releases, bug fixes for the release will follow the standard convention and be merged back into the appropriate release branch. Our release branches will follow the form: taint_model_verifier_<release number>. For release branches, names will be entirely lowercase.

We understand that bugs are a natural part of the development process and anticipate finding them ourselves as well as being informed by the client that bugs have been found. If the development team finds a bug, we will add an item to our Trello board so that it receives appropriate attention. When the client brings a bug to our attention, we will add it to our Trello Board and add an issue on GitHub to allow for external tracking.

We anticipate that we will find or be presented with bugs faster than we are able to fix them, so it will be important that we prioritize effectively. We will choose to focus most on those bugs which are most critical to our baseline goals; we will analyze bugs to determine which of our goals and risks are placed in danger as a consequence of the bug, and we will prioritize most highly bugs which most impact our highest priority goals and most substantial risks. We of course recognize that the client may express conflicting preferences for bug prioritization, so deviations from the plan are possible.

We specify three categories of severity for bugs and will assign each bug a severity; major bugs threaten the integrity of the system and its viability as a product; moderate bugs limit the functionality of the system but do not derail the system as a whole; and minor bugs have little to no practical impact. Examples of each are given in the table below:

Minor	Moderate	Major
Output contains correct information but is formatted suboptimally	Some instructions which freeze the system and are occasionally but infrequently randomly generated are not blocked by the filterer.	The system cannot run multiple instructions and, when multiple instructions are specified in a list, freezes, or repeatedly runs the first instruction in the list

Major bugs will be addressed first, followed by Moderate bugs, and finally Minor bugs. As we implement new features, we will need to decide whether to prioritize these features or existing bugs. Generally speaking, Major bugs will be prioritized over new features, as we have defined these bugs as critical to the integrity of our system; Minor bugs will not be prioritized above new features (unless, of course, they interfere with these features); and Moderate bugs and new features may be prioritized in either order. Prioritization of Moderate bugs and new features is likely to depend on client preference.

Regardless of who finds a particular bug, we will create a hotfix branch off of the branch containing the bug, and, when it is fixed, merge it back into the original branch. These hotfix branches will follow the form: username_mm/dd/yyy_BUG_<bugName>. In keeping with other branch names, usernames will be listed in lowercase and the bug name will be shortened and in camel case. The following table contains examples of bug fix branches:

Branch Name	Developer	Purpose
snidersa_3/16/2022_BUG_InstructionsNotChanging	Simon Snider	Fixing an issue where instructions were not switching during panda's emulation
menga_01/25/2022_BUG_reflexive_correlations_26_to_29	Andrew Meng	Fixing an issue in the correlation calculator that always included reflexive correlations for registers 26 through 29

Just as we anticipate bugs' being brought to our attention by the client, we anticipate reprioritization of requirements. When a new feature or requirement is introduced into the system, our approach will depend on the agreed upon timeframe. No new branches will be created immediately in the vast majority of cases; instead, additional features will be handled as feature branches in the usual way. If a new requirement is deemed sufficiently urgent by the client, it may be considered in a similar manner to a bug. In this case, a new branch will be created immediately, named as a feature branch. If the new requirement/feature is required on a past release, it will be merged directly into that release after being fully tested and after being merged into main.

In the event that a newly introduced requirement conflicts with a pre-existing requirement, additional communication with the client will be necessary. We will engage with the client and reconcile any inconsistencies until the new requirement has been formalized. At this point, we will integrate it into our schedule and follow the process documented above.