CS 639A Progress Report 2

Sumit Lahiri	Group No	Amit Kumar Sharma
19111274	1	20111012

Reinforcement Learning Stage Report : Pre-RL & RL Stage

After completing the **stage 1** work, we moved to the **stage 2** work. Our first task to start modifying and preparing the various programs and **scripts**, **manifests** & **libraries** needed for both the **OCCAM** tool and **Chisel** tool. These tools already had picked up some benchmark runs, so our first trial was to prepare according to the following benchamarks for the tools, namely **Chisel Tool benchmarks** , **OCCAM Tool benchmarks** & **Trimmer Tool benchmarks**.

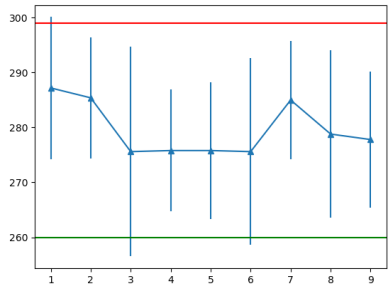
These benchmarks needed modification interms of running it against updated **gllvm**, **wllvm** & **llvm-10**. There were a few other tools and frameworks that we had to install and test in-order to run the debloating tools on the current benchamarks as listed above.

We devised that we can split the runs and the debloating process for comparision on three seperate pipelines namely, **chisel-tool**, **occam-tool** & **deepoccam-tool** pipeline. Our intension is to run each tool seperately on a single example and then compare them against the **GadgetSetAnalyser**. We list below the work we did on each pipeline for **Week-2** & **Week-3** of our development. We lag behind a bit in the **Week-4** activity, which we intend to mitigate in the last week hopefully.

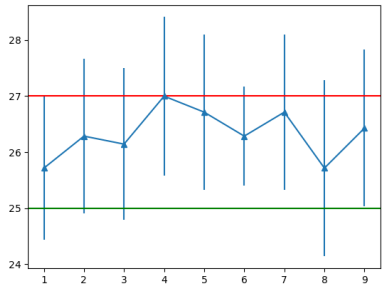
Pipeline : Chisel Tool

We share a brief information about the tool. **Chisel** tool works by learning a **policy** for **delta debugging** by **reinforcement learning** which guarantees **1-minimal P*** & $O(|P|^2)$ runtime. The abstraction is that a **markov decision process** is being used to model the **reinforcement learning** problem for meaningful **guidance** to learn the **policy** in a better way. All global declarations, variables, functions etc. are first reduced by the delta-debugging principle as state above and thereafter local variables, loop declarations and arguments to functions are optimized. After both the local and global level reductions are done, **Chisel** invokes a run of the **global level reduction** again and repeats the process continually until the **1-minimal P*** version of the program is found.

- Install and preparation work for **Chisel Tool benchmarks** and **wllvm**. The code was dated so we made the modifications to run and process it further.
- Fixed the **code**, `Makefile` & `script` files for running on our platform.
- We wrote specification for the script used by **Chisel** tool as in Paper 1 in terms of the three parts of the scripts namely `compile()`, `desired()` & `undesired()` for some of the benchmark runs.
- Generating a proper binary after reduction was difficult given we had some missing **linked shared objects** & **undefined symbols** issue. We resorted to building and modifiying **Chisel** tool and the benchmark examples in **docker containers** against the correct tools and setup for **GadgetSetAnalyser**



(a) Chisel Tool Run : Function 1 Example



(b) Chisel Tool Run : Function 2 Example

Figure 1: Chisel Tool runs on `two examples`. **Before** and **After** debloating.

- Prepare **GadgetSetAnalyser** to **ROP Gadgets** only. We need to plot the results properly and extend it to show the total number of unqiue gadgets during submission.
- **We have completed a basic run of our pipeline on Chisel Tool**

```
Running delta debugging - Size: 0
Start local reduction
Reduce main at ./test/function1/function1.c
Reduce File: ./test/function1/function1.c
Iteration 1 (Word: 4)
Start global reduction
Running delta debugging - Size: 0
Start local reduction
Reduce main at ./test/function1/function1.c
...
```

Final Results from the tool. We show a partial dump here.

```
=================================================
                Report
=================================================
   Total Time :                          1.1s
   Oracle Time :                         0.5s
   Learning Time :                       0.1s
   Global Success Ratio :          0% ( 0 / 0)
   Local Success Ratio :         100% ( 3 / 3)
   Functions (Original) :                   1
   Statements (Original) :                  4
   Functions (Reduced) :                    1
   Statements (Reduced) :                   2
```

Pipeline : OCCAM Tool

OCCAM is a whole-program partial evaluator for LLVM bitcode that aims at debloating programs and shared/static libraries running in a specific deployment context.

- Prepared the examples and the benchmark codes for **OCCAM** runs.
- Fixed the **code**, `Makefile` & `script` files for running on our platform.

- We had to refactor the code and modify it for **docker containers** run. Most of the work was to link it properly with it's dependencies and build it for the base level run.
- Running the tool against some examples in the **OCCAM** repository. We ran our modified tool in this case.
- Running the tool in following `none`, `aggressive`, `bounded` & `onlyonce` modes. We shall detail out the runs and the options in **report** we submit.
- We ran the **OCCAM** tool for run on **bzip2**, **httpd**, **tree** and many other programs as listed in the benchmarks.
- Compared to **Chisel** as we ran **OCCAM** on some sample `C` & `C++` code.
- Refactored the code to support the new version for **sea-dsa** for the `--use-pointer-analysis` option needed to do futher `dce` on the code and also to support `--ai-dce`

A sample run of the tool where we show that **OCCAM** tool has specialized some of the **call-sites**

```
[23/11/2020 19:18:51] STARTED runbench
[23/11/2020 19:18:51] RUN make on tree
Running slash with options ['./build.sh', '--disable-inlining',
    '--enable-config-prime']
[23/11/2020 19:18:53] RUN ./build.sh --disable-inlining --enable-config-prime on
    tree
[23/11/2020 19:18:55] FINISHED runbench

Program Reduction: (B:before and A:after OCCAM with --disable-inlining
    --enable-config-prime)

Program B Fun A Fun %Fun Red B Ins  A Ins  %Ins Red B Mem Ins A Mem Ins %Mem Ins Red
tree      52    32       38  7442   4110        44      1658       810          51
```

Results for `GNU Tree` program on **OCCAM** tool on **llvm-10** after required modifications & **successful** build. We show a partial dump here.

```
Statistics for tree before specialization
[CFG analysis]
52 Number of functions
0 Number of specialized functions
0 Number of bounced functions added by devirt
1458 Number of basic blocks
7442 Number of instructions
1051 Number of direct calls
845 Number of external calls
15 Number of indirect calls
0 Number of unknown calls
43 Number of loops
0 Number of bounded loops
[Memory analysis]
1944 Number of memory instructions
286 Statically safe memory accesses
1658 Statically unknown memory accesses

Statistics for tree after specialization
[CFG analysis]
40 Number of functions
0 Number of specialized functions
0 Number of bounced functions added by devirt
1845 Number of basic blocks
8957 Number of instructions
1257 Number of direct calls
1167 Number of external calls
```

```
15 Number of indirect calls
0 Number of unknown calls
66 Number of loops
0 Number of bounded loops
[Memory analysis]
2222 Number of memory instructions
348 Statically safe memory accesses
1874 Statically unknown memory accesses
```

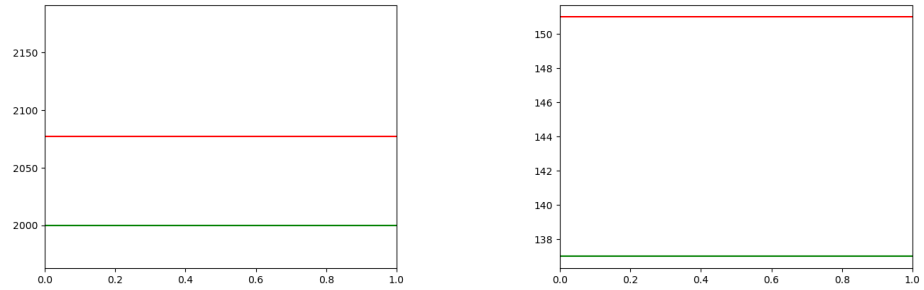
Results for `GNU bzip2` program on **OCCAM** tool on **llvm-10** after required modifications & **successful** build.

```
Statistics for bzip2 before specialization
[CFG analysis]
108 Number of functions
0 Number of specialized functions
0 Number of bounced functions added by devirt
3223 Number of basic blocks
26718 Number of instructions
678 Number of direct calls
304 Number of external calls
0 Number of assembly calls
20 Number of indirect calls
0 Number of unknown calls
161 Number of loops
0 Number of bounded loops
[Memory analysis]
12589 Number of memory instructions
8304 Statically safe memory accesses
4285 Statically unknown memory accesses

Statistics for bzip2 after specialization
[CFG analysis]
94 Number of functions
0 Number of specialized functions
0 Number of bounced functions added by devirt
2334 Number of basic blocks
19877 Number of instructions
672 Number of direct calls
293 Number of external calls
0 Number of assembly calls
20 Number of indirect calls
0 Number of unknown calls
180 Number of loops
18 Number of bounded loops
[Memory analysis]
5187 Number of memory instructions
486 Statically safe memory accesses
4701 Statically unknown memory accesses
```

Pipeline : DeepOCCAM Tool

DeepOCCAM does specialization and optimizations with the help of **OCCAM** tool. The **DeepOCCAM** tool is different from **OCCAM** in the sense that **DeepOCCAM** does deep learning implemented in **PYTORCH** to learn a policy. The model also keeps information of the decisions made on each iteration to update the policy via `def push_to_memory(self, memory)`. We found a **half-implemented code** repository which belongs to one of the authors for **DeepOCCAM** paper. The tool doesn't link, build or run on the current platform that we are using. It gave us insights on implementing some of the parts in the code that we are developing as an extension on **OCCAM** to develop **DeepOCCAM**.



(a) OOCAM Run : `bzip2`

(b) OOCAM Run : `module` example

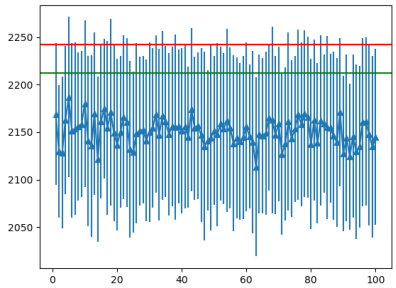
Figure 2: OCCAM Tool runs on `bzip2` & `module`. **Before** and **After** debloating

- Modified code in base **OCCAM** tool to collect **RL** related features and added support for **gRPC** server which can run and communicate with the **PyTorch** RL Model running as client. **(We are facing linking issues here.)**
- Implemented a **deep reinforcement learning model** from the insights we got from the **half-implemented code**.
- We setup a **docker** based **deepoccam combined-tools** container pipeline to build, run and debloat an example **code repository**. The **training** part happened outside the docker, we just saved and extracted the model for later use in **deepoccam-combined-tools** container.
- We had to modify the way the tool was plotting and processing some the feature vectors. We used `Adam Optimization`, `ReLU` functions, `Softmax`, `torch.nn GRU` neural net for `inst2vec` & `Linear fully-connected` layers etc. for implementing the `ML` part of **DeepOCCAM** tool.
- We are yet to complete the training phase over **gRPC** connection where the **OCCAM** tool runs and gives the data for the `RL Features`. We are currently running **OCCAM** tool in **non-gRPC** mode and training the tool based on some `metadata.json` files we got in the **half-implemented code** repository.
- The **half-implemented code** repository really helped us in correcting and implementing some of the design decisions we made in the early implementation of the **DeepOCCAM** tool from **OCCAM**.
- As before, we are developing from **OCCAM** tool repository to build **DeepOCCAM** after gaining some insights and **metadata** files from **half-implemented code** we got.

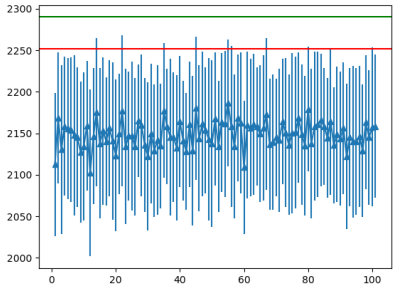
Metrics Testing : COP, ROP & JOP Gadgets

Call Oriented programming (COP),**Jump Oriented programming (JOP)** and **Return Oriented Programming (ROP)** are **computer security exploitation** techniques. The hacker uses the binary instructions to combine some sort of short sequences of instructions which are commonly called **gadgets** using return section code in the stack and make **unwanted** stack sections directly **executable**. The gadgets like COP, ROP & JOP **hijack** the actual control flow of the program and once the control is disturbed, hacker can make use of this **opportunity** to attack the system. Therefore, it is necessary to keep these gadgets as minimum as possible. While **debloating** the program **reduces** the number of gadgets, it is still not guaranteed that the attack is prevented but it **minimizes** the attack surface to a certain extent.

It is extremely important to keep track of the metrics of **gadgets** as these metrics when used to learn a RL policy will help in maximizing the rewards which in turn reduces the **code**



(a) DeepOOCAM Run 20 : `bzip2` training



(b) DeepOOCAM Run 50 : `bzip2` training

Figure 3: DeepOCCAM Tool runs on `bzip2`. **Before** and **After** debloating

attack surface. So while testing **DeepOCCAM**, these gadgets counts need to be tracked. We use the **number** of **gadgets** we get by running each tool namely **Chisel**, **trimmer**, **OCCAM** or our implementation of **DeepOCCAM** as a **reasonably important** metric of comparision.

A word on **Trimmer**, **OCCAM** also uses partial evaluation concept like **Trimmer** for specializing applications but it is less aggressive compared to the **Trimmer** because it does not include `loop unrolling` and `constant propagation` for optimizations.

Pending Works

We have to complete some parts of the **DeepOCCAM** tool in the **RL Model** part. **We are yet to run the three tools on common examples to get enough data for a good comparision report and smooth running implementation**. We may have to skip the **gRPC** based implementation upon a **time crunch**. Building, fixing, refactoring and developing the tools for a smooth run took **more time than we actually expected**.

Code & Docker Links for Tools

- OCCAM Tool : **https://github.com/lahiri-phdworks/OCCAM**
- OCCAM Test & Benchmarks : **https://github.com/SRI-CSL/OCCAM-Benchmarks**
- Chisel-Bench : **https://github.com/lahiri-phdworks/chisel-bench**
- Chisel Tool : **https://github.com/lahiri-phdworks/chisel**
- Inst2Vec : **https://github.com/lahiri-phdworks/ncc**
- Binary GSA Testing Tool : **https://github.com/michaelbrownuc/GadgetSetAnalyzer**
- Docker Images (recent ones) : **https://hub.docker.com/u/prodrelworks**