UNIVERSITY OF NAPLES FEDERICO II

SCHOOL OF POLYTECHNIC AND BASIC SCIENCES

DEPARTMENT OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGIES

Master's degree in computer engineering

ANALYSIS OF THE FLASHSYN TOOL

Student:

Migliaccio Luca

Summary

Section 1: Introduction

Objective

The **objective** of this exercise is to reproduce one or more rows of **Table 3** for paper "*FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation (ICSE 2024)*", given the code and data in Github repository (*https://github.com/FlashSyn-Artifact/FlashSyn-Artifact-ICSE24*).

						FlashSyn-poly			FlashSyn-inter			Precise
Benchmark	**AC**	**AP**	**GL**	**GP**	**IDP**	**TDP**	**Profit**	**Time**	**TDP**	**Profit**	**Time**	**Profit**
bZx1	3	3	2	1194	5192	5849	2392	422	6373	2302†	441	cs
Harvest_USDT	4	4	4	338448	8000	9325	110139†	670	10289	86798†	7579	cs
Harvest_USDC	4	4	4	307416	8000	8912	59614†	677	10914	110051†	8349	cs
Eminence	4	4	5	1674278	8000	8780	1507174	1191	8104	/	/	1606965
ValueDeFi	6	6	6	8618002	12000	19975	8378194†	4691	15758	6428341†	11089	cx
CheeseBank	8	3	8	3270347	2679	2937	1946291†	4391	2715	1101547†	10942	2816762†
Warp	6	3	6	1693523	6000	6000	2773345†	1164	6000	/	/	2645640†
bEarnFi	2	2	4	18077	4000	4854	13770	470	4652	12329	688	13832
AutoShark	8	3	8	1381	2753	2753	1372†	5484	2753	/	/	cx
ElevenFi	5	2	5	129741	4000	4070	129658	409	4326	85811	898	cx
ApeRocket	7	3	6	1345	6000	6402	1333†	733	6235	1037†	3238	cs
Wdoge	5	1	5	78	2000	2001	75	272	2080	75	289	75
Novo	4	2	4	24857	4000	4164	20210	702	4031	23084	861	cx
OneRing	2	2	2	1534752	4000	4710	1814882	585	4218	1942188	367	cx
Puppet	3	3	2	89000	6000	6301	89000†	1203	6452	87266†	1238	89000†
PuppetV2	4	3	3	953100	4491	4836	747799†	2441	5061	362541†	2835	647894†
						Solved:16/18 **Avg. Time**: 1594			**Solved**:13/18 **Avg. Time**: 3754			

System environment

The tests were executed on a personal computer with these features:

- **OS:** Windows 11 Home (Version 24H2, Build 26100.3915)
- **Processor:** AMD Ryzen 7 5825U with Radeon Graphics, 2.00 GHz
- **RAM:** 16 GB
- **Architecture:** 64-bit OS, x64-based processor

To ensure compatibility with FlashSyn, which is designed for Linux-based environments, **Windows Subsystem for Linux (WSL)** was used, with **Ubuntu 22.04** serving as the Linux distribution within the subsystem.

Steps to Execute FlashSyn in Docker

To verify the correct functioning of FlashSyn, it was executed in a Docker container following these steps:

1. **Install Docker** and set up **Ubuntu via WSL** to allow Linux-based container execution on Windows.
2. Pull the FlashSyn Docker image → *sudo docker pull zhiychen597/flashsyn:latest*

```
lm@lucam:~/project_FlashSyn$ sudo docker pull zhiychen597/flashsyn:latest
latest: Pulling from zhiychen597/flashsyn
16ea0e8c8879: Pull complete
50024b0106d5: Pull complete
ff95660c6937: Pull complete
9c7d0e5c0bc2: Pull complete
29c4fb388fdf: Pull complete
8659dae93050: Pull complete
1da0ab556051: Pull complete
e92ae9350d4a: Pull complete
c648cb7fc575: Pull complete
ea3ee54b8ae5: Pull complete
821dd0780458: Pull complete
83f83b9c3793: Pull complete
3f4d44b79139: Pull complete
c26215707004: Pull complete
d65745795ba1: Pull complete
9bd0f1e6ed4e: Pull complete
a1876a2ebb87: Pull complete
bbb9fc2d3312: Pull complete
Digest: sha256:21fca8cdef66d092825239f39fa4b6f193763e71b70139cfdccd7d1232a961c8
Status: Downloaded newer image for zhiychen597/flashsyn:latest
docker.io/zhiychen597/flashsyn:latest
lm@lucam:~/project_FlashSyn$
```

3. **Create a Docker volume** to persist output data (such as logs), which is necessary to avoid data loss when the container is stopped → *sudo docker volume create FlashSyn-Data-Reproduce*

```
lm@lucam:~/project_FlashSyn$ sudo docker volume create FlashSyn-Data-Reproduce
FlashSyn-Data-Reproduce
lm@lucam:~/project_FlashSyn$
```

4. **Run the Docker container** with the volume mounted to save results in a persistent location → *sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/flashsyn:latest bash*

```
lm@lucam:~/project_FlashSyn$ sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/
flashsyn:latest bash
root@42b03ffe505b:/FlashSyn#
```

5. Prepare the environment inside the container → *. ~/.bashrc*
   forge -V

```
root@42b03ffe505b:/FlashSyn# . ~/.bashrc
root@42b03ffe505b:/FlashSyn# forge -V
forge 0.2.0 (6fc06c5 2024-01-05T02:43:33.3154492792)
```

Section 2: Initial Functionality Check of FlashSyn via Baseline Test Execution

In this chapter, we perform a test to verify the correct functionalities of FlashSyn. In addition, the process of interpreting the results is explained.

Test Execution

The steps followed are:

1. **Run the FlashSyn test** (execute the baseline test from inside the Docker container) → *chmod +x ./runTest.sh && ./runTest.sh*

```
root@42b03ffe505b:/FlashSyn# ls
Benchmarks              LICENSE     README3.md            flashsyn.tar      runRQ1.sh  runRQ4.sh      src
Dockerfile              README.md   Results-Expected      paper             runRQ2.sh  runTest.sh     temp.txt
HOW-TO-READ-DATA-LOG.md README2.md  Results-To-Reproduce  requirements.txt  runRQ3.sh  settings.toml
root@42b03ffe505b:/FlashSyn# chmod +x ./runTest.sh && ./runTest.sh
========= Current Date and Time: Wed Apr 30 08:15:45 UTC 2025 =========
running FlashSyn-precise baseline for bEarnFi...
bEarnFi (precise) done.
================================================================================================================
========================================== ALL COMPLETE ==========================================
================================================================================================================
root@42b03ffe505b:/FlashSyn#
```

2. **Verify the test result** (by inspecting the log file) → *cat Results-To-Reproduce/FlashSynData/precise/bEarnFi_precise.txt*

```
root@42b03ffe505b:/FlashSyn# cat Results-To-Reproduce/FlashSynData/precise/bEarnFi_precise.txt
/FlashSyn
Deposit number of points:
 skip
EmergencyWithdraw number of points:
0
Check Contract:         Deposit, EmergencyWithdraw  time: 0.0558171272277832
The optimizer takes 0.06953740119934082 seconds
best para: [6875000.3125] best profit: 9.313225746154785e-10
Optimization terminated successfully.    Next only show the first 5/7 profitable solutions
[4765625]         estimated profit is, 9.313225746154785e-10
[5156250]         estimated profit is, 9.313225746154785e-10
[6171875]         estimated profit is, 9.313225746154785e-10
[6406250]         estimated profit is, 9.313225746154785e-10
[6562500]         estimated profit is, 0.0

Check Contract:         Deposit, EmergencyWithdraw, Deposit, EmergencyWithdraw  time: 0.15732431411743164
The optimizer takes 0.046114444732666016 seconds
best para: [7109375.2890625 7109375.2890625] best profit: 11550.858996806666
Optimization terminated successfully.    Next only show the first 1/1 profitable solutions
[7109375, 7109375]        estimated profit is, 11550.858996806666
[7095156, 7095156]        estimated profit is, 11506.077953413129
```

Last lines of the log file:

```
================================================================================
======================== End of Synthesis, time in total:  26.944101333618164 s ===============
================================================================================
========================== Now shows the answers: ============================================
======================== End of Answers    ====================================
==================== Best Profit  19.200000000000003  ==============================
```

Note: The *best profit* value obtained differs from the expected one (as indicated in the README). This may be due to differences in the **hardware resources** of the machine used.

How to read a log file

The **log file** generated by experiments is a **.txt file**. It contains all the execution details of FlashSyn for a specific benchmark, such as:

- the collected data
- the tested attack candidates
- the estimated and actual profits

- the refinement phases
- the best attack found

Start of the file → FlashSyn shows how many *data points* were initially used for each contract function (e.g., swap, deposit, etc.). This reflects the *configuration*, such as: “200 per action.”

From line 6 onward → You see a list of *candidate attack vectors* generated by FlashSyn. For each one, the optimizer tries different parameters to estimate *theoretical profit*.

From line 26 onward → FlashSyn simulates the attack on the forked blockchain and calculates the actual (real) profit—not just the estimated one. These are real executions, not just predictions.

Lines 217–223 → This is a summary of the best attack found up to that point:

- which sequence of actions
- which parameters
- how much actual profit it generated

From round 1 onward → Refinement begins:

- If the estimated result ≠ the actual result → it's a counterexample
- FlashSyn collects more data at that point and improves the approximation

End of the file – Lines 1823–1824 → At the very end, there's a total summary:

- All the tested attacks
- Those that yielded positive profit
- The best one overall, with the exact parameters that produced it.

Final note → Due to internal dataset handling, some files are labeled “in reverse”:

- Harvest_USDT.txt actually refers to the USDC case
- And vice versa

It’s just a naming issue in the files—the data itself is correct.

Section 3: Research Questions

Our goal is to respond to first and second research questions indicated by paper (**RQ1** and **RQ2**). Before carrying out the experiments, it was necessary to make some **considerations**:

- **Note 1**: the **shgo solver** is *non-deterministic* and may adopt different search strategies depending on the hardware. This means results may slightly vary from one machine to another. What matters is whether *an attack with a positive profit* is found—not the exact profit value.
- **Note 2**: the original authors run FlashSyn using up to *18 parallel processes*, but the provided Docker version limits it to a *single process*. This can make FlashSyn slower and may result in slightly worse performance.
- **Note 3**: The Docker image has been tested only on *Ubuntu AMD-based machines*. There may be *compatibility issues* on macOS or Windows systems, especially ARM-based platforms.

Subsection 3.1: RQ1

RQ1 says: *"How effective is FlashSyn in synthesizing flash loan attack vectors?"*. To evaluate how well FlashSyn performs using **2000 initial data points combined with counterexample-driven refinement**, we run the RQ1 experiments on **three selected benchmarks: bEarnFi, Wdoge** and **CheeseBank**. *bEarni* and *Wdoge* were chosen based on the **execution times** reported in **Table 3** of the original paper-specifically, they are among the fastest to run for both synthesis techniques (*polynomial regression* and *interpolation*). In addition, *CheeseBank* was chosen to conduct an experiment on *a benchmark where FlashSyn's results include at least one attack vector that differs from the ground truth*.

bEarnFi and Wdoge: Execution

We launched the Docker container with mounted volume for persistent result storage → *sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/flashsyn:latest bash*

```
lm@lucam:~/project_FlashSyn$ sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/
flashsyn:latest bash
[sudo] password for lm:
root@517600c32a9d:/FlashSyn#
```

Inside the container, apply shell configurations and grant execution permissions to the script → *. ~/.bashrc*

chmod +x ./runRQ1.sh

```
root@517600c32a9d:/FlashSyn# . ~/.bashrc
root@517600c32a9d:/FlashSyn# chmod +x ./runRQ1.sh
```

Each of the following commands runs FlashSyn on one benchmark using both approximation techniques (poly and interpolation):

- For **bEarnFi** → *./runRQ1.sh bEarnFi*

```
root@ec0ef2b40145:/FlashSyn# ./runRQ1.sh bEarnFi
========= Current Date and Time: Thu May  1 10:08:55 UTC 2025 =========
running FlashSyn-inter with 2000 initial datapoints with counterexample driven loops for bEarnFi...
bEarnFi done.
Benchmarks: 1/32 finished, 31/32 to do
========= Current Date and Time: Thu May  1 10:16:50 UTC 2025 =========
running FlashSyn-poly with 2000 initial datapoints with counterexample driven loops for bEarnFi...
bEarnFi done.
Benchmarks: 2/32 finished, 30/32 to do
================================================================================================
========================================= ALL COMPLETE =========================================
================================================================================================
```

- For **Wdoge** →

```
root@ec0ef2b40145:/FlashSyn# ./runRQ1.sh Wdoge
========= Current Date and Time: Thu May  1 10:25:54 UTC 2025 =========
running FlashSyn-inter with 2000 initial datapoints with counterexample driven loops for Wdoge...
Wdoge done.
Benchmarks: 1/32 finished, 31/32 to do
========= Current Date and Time: Thu May  1 10:27:25 UTC 2025 =========
running FlashSyn-poly with 2000 initial datapoints with counterexample driven loops for Wdoge...
Wdoge done.
Benchmarks: 2/32 finished, 30/32 to do
================================================================================================
========================================== ALL COMPLETE ==========================================
================================================================================================
```

Each execution performs:

- **FlashSyn-poly** (using polynomial regression)
- **FlashSyn-inter** (using nearest-neighbor interpolation)
- Both with **2000 initial data points per action**, and with **refinement loops based on counterexamples**

Note: Although the script runRQ1.sh is typically used to run **32 executions** (*16 benchmarks × 2 methods*), in our case it stops at 2/32 because we explicitly provide only one benchmark per run.

bEarnFi and Wdoge: Analysis of Results

After completing the executions for both benchmarks, we run the following script to analyze the output and extract the relevant data for comparison → *python3 Results-To-Reproduce/RQ1.py*:

```
root@517600c32a9d:/FlashSyn# python3 Results-To-Reproduce/RQ1.py
          ||   FlashSyn-poly    ||   FlashSyn-inte  ||
benchmark GP IDP TDP Profit Time TDP Profit Time
bZx1 1194 - -
Harvest_USDT 338448 - -
Harvest_USDC 307416 - -
Eminence 1674278 - -
ValueDeFi 8618002 - -
CheeseBank 3270347 - -
Warp 1693523 - -
Yearn 56924  ----------------------------------------
InverseFi 2515606  ----------------------------------------
bEarnFi 18077 4000 4000 / / 4000 / /
AutoShark 1381 - -
ElevenFi 129741 - -
ApeRocket 1345 - -
Wdoge 78 2000 2000 / / 2000 / /
Novo 24857 - -
OneRing 1534752 - -
Puppet 89000 - -
PuppetV2 953100 - -
================================================================================================
Solved(poly):  0 out of 18  Traceback (most recent call last):
  File "Results-To-Reproduce/RQ1.py", line 267, in <module>
    main()
  File "Results-To-Reproduce/RQ1.py", line 254, in main
    print("Avg Time:", int(sum(Time_poly)/len(Time_poly)), end = "  ")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The script:

- Reads the *log files generated* (*_poly.txt, *_inte.txt)
- Extracts *profit*, execution *time*, and *number of data points*
- Prints a *summary* in a table format like the one shown in the paper

As observed, there are **some errors**: the Time_poly list contains string values ('/') in addition to integers. As a result, the *sum()* function fails. To fix this issue, it was necessary to **modify the RQ1.py script**, specifically by reducing the list of benchmarks considered.

```python
def getProfitinHistory(benchmark: str):
    profit = 0
    # TEST 1
    if benchmark == "bEarnFi":
        profit = 18077.148053847253
    elif benchmark == "Wdoge":
        profit = 78

    return int(profit)

def main():



    method = 0 # 0 for interpolation
               # 1 for polynormial

    benchmarkList = ['bEarnFi', 'Wdoge'] # TEST 1
```

and

```python
# TEST 1
print("Profit_inte", Profit_inte)
print("len: ", len(Profit_inte))
print("Profit_poly", Profit_poly)
print("len: ", len(Profit_poly))
print("Profit_his", Profit_his)
print("len: ", len(Profit_his))
print("Time_inte", Time_inte)
print("len: ", len(Time_inte))
print("Time_poly", Time_poly)
print("len: ", len(Time_poly))
```

We then move the script into the Docker container (based on its ID):

```
lm@lucam:~/project_FlashSyn/script_test$ docker cp RQ1_test1.py ec0ef2b40145:/FlashSyn/Results-To-Reproduce
Successfully copied 11.3kB to ec0ef2b40145:/FlashSyn/Results-To-Reproduce
lm@lucam:~/project_FlashSyn/script_test$
```

and we verify that the operation was successful by checking the contents of the file inside the container with:

```
root@ec0ef2b40145:/FlashSyn/Results-To-Reproduce# ls
FlashFind+FlashSynData  RQ1.py        RQ2.py             RQ3_NormProfit+Solved.py  RQ4.py
FlashSynData            RQ1_test1.py  RQ3_Datapoints.py  RQ3_Time.py
root@ec0ef2b40145:/FlashSyn/Results-To-Reproduce#
```

Execution → *python3 Results-To-Reproduce/RQ1_test1.py*

```
root@ec0ef2b40145:/FlashSyn# python3 Results-To-Reproduce/RQ1_test1.py
          ||   FlashSyn-poly    ||   FlashSyn-inte  ||
benchmark GP IDP TDP Profit Time TDP Profit Time
bEarnFi 18077 4000 4000 / / 4000 / /
Wdoge 78 2000 2000 / / 2000 / /
Profit_inte [0, 0]
len:  2
Profit_poly [0, 0]
len:  2
Profit_his [18077, 78]
len:  2
Time_inte []
len:  0
Time_poly ['/', '/']
len:  2
================================================================================
Solved(poly):  0 out of 18  Traceback (most recent call last):
  File "Results-To-Reproduce/RQ1_test1.py", line 232, in <module>
    main()
  File "Results-To-Reproduce/RQ1_test1.py", line 219, in main
    print("Avg Time:", int(sum(Time_poly)/len(Time_poly)), end = "  ")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The **error** displayed (*TypeError: unsupported operand type(s) for +: 'int' and 'str'*) occurs because the Time_poly list contains strings (e.g., '/') instead of only integers. When the script tries to calculate the average using *sum(Time_poly)*, python crashes.

We therefore modify the Python script again and rerun it:

```
print("================================================================================================")
NumOfSolved_poly = 0
for profit in Profit_poly:
    if profit > 0:
        NumOfSolved_poly += 1
print("Solved(poly): ", NumOfSolved_poly, "out of 18", end = "  ")
filtered_Time_poly = [t for t in Time_poly if isinstance(t, int)]
if filtered_Time_poly:
    print("Avg Time:", int(sum(filtered_Time_poly)/len(filtered_Time_poly)), end = "  ")
else:
    print("Avg Time: /", end = "  ")

NumOfSolved_inte = 0
for profit in Profit_inte:
    if profit > 0:
        NumOfSolved_inte += 1
print("Solved(inte): ", NumOfSolved_inte, "out of 18", end = "  ")
filtered_Time_inte = [t for t in Time_inte if isinstance(t, int)]
if filtered_Time_inte:
    print("Avg Time:", int(sum(filtered_Time_inte)/len(filtered_Time_inte)), end = "  \n")
else:
    print("Avg Time: /", end = "  \n")
```

```
root@ec0ef2b40145:/FlashSyn# python3 Results-To-Reproduce/RQ1_test1.py
          ||   FlashSyn-poly    ||   FlashSyn-inte  ||
benchmark GP IDP TDP Profit Time TDP Profit Time
bEarnFi 18077 4000 4000 / / 4000 / /
Wdoge 78 2000 2000 / / 2000 / /
Profit_inte [0, 0]
len:  2
Profit_poly [0, 0]
len:  2
Profit_his [18077, 78]
len:  2
Time_inte []
len:  0
Time_poly ['/', '/']
len:  2
================================================================================================
Solved(poly):  0 out of 18  Avg Time: /  Solved(inte):  0 out of 18  Avg Time: /
root@ec0ef2b40145:/FlashSyn#
```

Apparently, it seems that no attacks were found. That's why the alarm threshold was set to 40 (*profit > 40*). If we analyze the log files for both benchmarks (clearly considering the **2000+X folder**, since we performed the analysis with *2000 initial data points and counterexample-driven refinement*):

For bEarnFi (first method) → *cat Results-To-Reproduce/FlashSynData/2000+X/bEarnFi_inte.txt*

```
Deposit number of points:
2000
EmergencyWithdraw number of points:
2000
 ================================================================================
 ======================== End of Synthesis, time in total:  460.6286678314209 s ==============
 ================================================================================
 ======================== Now shows the answers: ================================================
 ======================== End of Answers    ======================================
 =================== Best Profit  16.0  ============================
```

Functions tested: Deposit and EmergencyWithdraw

Number of data points per function:

- Deposit: 2000
- EmergencyWithdraw: 2000

Total synthesis time: ~460.63 seconds (about 7 minutes and 41 seconds)

Best profit found: 16.0

Interpretation: FlashSyn ran the benchmark correctly, simulating various interaction scenarios between Deposit and EmergencyWithdraw. It identified a potential attack with a profit of 16.0 (units on the same log scale, typically in USD or equivalent assets). The fact that there is a positive profit implies that there is a sequence of actions that can exploit the logic of the contract to make a profit.

For bEarnFi (second method) → *cat Results-To-Reproduce/FlashSynData/2000+X/bEarnFi_poly.txt*

```
Deposit number of points:
2000
EmergencyWithdraw number of points:
2000
 ================================================================================
 ======================== End of Synthesis, time in total:  50.22027611732483 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
 ======================== End of Answers    ====================================
 =================== Best Profit  5.6000000000000005  ============================
```

For Wdoge (first method) → *cat Results-To-Reproduce/FlashSynData/2000+X/Wdoge_inte.txt*

```
SwapWBNB2Wdoge number of points:
 skip
TransferWdoge number of points:
 skip
PancakePairSkim number of points:
2000
PancakePairSync2 number of points:
 skip
SwapWdoge2WBNB number of points:
 skip
 ================================================================================
 ======================== End of Synthesis, time in total:  89.1891930103302 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
 ======================== End of Answers    ====================================
 =================== Best Profit  5.0  ============================
```

For Wdoge (second method) → *cat Results-To-Reproduce/FlashSynData/2000+X/Wdoge_poly.txt*

```
SwapWBNB2Wdoge number of points:
 skip
TransferWdoge number of points:
 skip
PancakePairSkim number of points:
2000
PancakePairSync2 number of points:
 skip
SwapWdoge2WBNB number of points:
 skip
 ================================================================================
 ======================== End of Synthesis, time in total:  30.672540426254272 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
 ======================== End of Answers    ====================================
 =================== Best Profit  4.2  ============================
```

I tested the RQ1 script again, this time changing the threshold (**> 0**), to see if and how the results changed. Once I modified the python script (*RQ1_test2.py*) and moved it to the container:

```
if globalbestProfit > 0: # TEST 2
    print(int(globalbestProfit), end = " ")
    if method == 1:
        Profit_poly.append(int(globalbestProfit))
    elif method == 0:
        Profit_inte.append(int(globalbestProfit))
else:
    print("/", end = " ")
    if method == 1:
        Profit_poly.append(0)
    elif method == 0:
        Profit_inte.append(0)

if Time > 0:
    if Time < 10000:
        if globalbestProfit > 0: # TEST 2
```

```
lm@lucam:~/project_FlashSyn/script_test$ docker cp RQ1_test2.py ec0ef2b40145:/FlashSyn/Results-To-Reproduce
Successfully copied 12.3kB to ec0ef2b40145:/FlashSyn/Results-To-Reproduce
lm@lucam:~/project_FlashSyn/script_test$
```

The **result** is:

```
root@ec0ef2b40145:/FlashSyn# python3 Results-To-Reproduce/RQ1_test2.py
          ||   FlashSyn-poly    ||   FlashSyn-inte  ||
benchmark GP IDP TDP Profit Time TDP Profit Time
bEarnFi 18077 4000 4000 5 81 4000 16 491
Wdoge 78 2000 2000 4 65 2000 5 124
Profit_inte [16, 5]
len:  2
Profit_poly [5, 4]
len:  2
Profit_his [18077, 78]
len:  2
Time_inte [491, 124]
len:  2
Time_poly [81, 65]
len:  2
================================================================================
Solved(poly):  2 out of 18  Avg Time: 73  Solved(inte):  2 out of 18  Avg Time: 307
root@ec0ef2b40145:/FlashSyn#
```

Note: the observed results are the same as those previously displayed with the "*cat*" command.

Conclusion: FlashSyn synthesized an attack with positive profit for both benchmarks, indicating a possible weakness in the contracts analyzed. However, there's a **problem**: **IDP = TDP = 4000** or **2000** → means no new points were added during refinement. FlashSyn was based only on the initial points.

Analyzing the log files (for both benchmarks and both methods), the **refinement did not start** because no concrete executable counterexamples were generated. Although FlashSyn estimated high profits, no concrete attacks were successful, so:

- No new points can be added
- The model cannot be corrected (no real feedback)
- FlashSyn stops after a few rounds.

One possible solution might be to *reduce the number of initial points* (e.g. 1000), so that the initial coverage is lower (more chance of triggering refinement).

In addition: there is a **precise folder** in the **FlashSynData folder**: it is used to store the *output* generated by FlashSyn *without using approximations* (such as interpolation or polynomial regression), but with a more accurate evaluation of the smart contract functions. It serves as a **baseline** comparison to evaluate how close the results obtained by approximate methods (interpolation, polynomial) are. It is used for the **RQ2 experiment**, which tests the effectiveness of FlashSyn in "*precise*" mode.

CheeseBank: Execution

We launched the Docker container with mounted volume for persistent result storage → *sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/flashsyn:latest bash*

```
lm@lucam:~/project_FlashSyn$ sudo docker run -it -v FlashSyn-Data-Reproduce:/FlashSyn/Results-To-Reproduce/ zhiychen597/
flashsyn:latest bash
[sudo] password for lm:
root@ebfbe281b202:/FlashSyn# ls
Benchmarks                LICENSE      README3.md            flashsyn.tar      runRQ1.sh  runRQ4.sh      src
Dockerfile                README.md    Results-Expected      paper             runRQ2.sh  runTest.sh     temp.txt
HOW-TO-READ-DATA-LOG.md   README2.md   Results-To-Reproduce  requirements.txt  runRQ3.sh  settings.toml
```

Inside the container, apply shell configurations and grant execution permissions to the script → *. ~/.bashrc*

chmod +x ./runRQ1.sh

```
root@ebfbe281b202:/FlashSyn# . ~/.bashrc
root@ebfbe281b202:/FlashSyn# chmod +x ./runRQ1.sh
```

We run FlashSyn with both approximation techniques (poly and interpolation)→ *./runRQ1.sh CheeseBank*:

```
root@ebfbe281b202:/FlashSyn# ./runRQ1.sh CheeseBank
========= Current Date and Time: Sat May  3 10:34:32 UTC 2025 =========
running FlashSyn-inter with 2000 initial datapoints with counterexample driven loops for CheeseBank...
./runRQ1.sh: line 28:    12 Killed                  timeout 10800s python3.7 src/main.py $case 0 1 2000 > ./$dir/${case}
_inte.txt
CheeseBank done.
Benchmarks: 1/32 finished, 31/32 to do
========= Current Date and Time: Sat May  3 13:09:26 UTC 2025 =========
running FlashSyn-poly with 2000 initial datapoints with counterexample driven loops for CheeseBank...
CheeseBank done.
Benchmarks: 2/32 finished, 30/32 to do
================================================================================================
========================================= ALL COMPLETE =========================================
================================================================================================
root@ebfbe281b202:/FlashSyn#
```

The execution performs:

- **FlashSyn-poly** (using polynomial regression)
- **FlashSyn-inter** (using nearest-neighbor interpolation)
- With **2000 initial data points per action**, and with **refinement loops based on counterexamples**

Note: Although the script runRQ1.sh is typically used to run **32 executions** (*16 benchmarks × 2 methods*), in our case it stops at 2/32 because we explicitly provide only one benchmark per run.

CheeseBank: Analysis of Results

First, we modified the pyhton script RQ1.py (**RQ1_CheeseBank.py**):

```python
def main():



    method = 0 # 0 for interpolation
               # 1 for polynormial

    benchmarkList = ['CheeseBank'] # TEST
```

We then move the script into the Docker container (based on its ID):

```
lm@lucam:~/project_FlashSyn/script_test$ docker cp RQ1_CheeseBank.py ebfbe281b202:/FlashSyn/Results-To-Reproduce
Successfully copied 12.3kB to ebfbe281b202:/FlashSyn/Results-To-Reproduce
lm@lucam:~/project_FlashSyn/script_test$
```

Execution → *python3 Results-To-Reproduce/RQ1_CheeseBank.py*

```
root@ebfbe281b202:/FlashSyn# python3 Results-To-Reproduce/RQ1_CheeseBank.py
          ||   FlashSyn-poly    ||   FlashSyn-inte  ||
benchmark GP IDP TDP Profit Time TDP Profit Time
CheeseBank 3270347 2679 2679 / / 2679 / /
Profit_inte [0]
len:  1
Profit_poly [0]
len:  1
Profit_his [3270347]
len:  1
Time_inte []
len:  0
Time_poly ['/']
len:  1
================================================================================================
Solved(poly):  0 out of 18  Avg Time: /  Solved(inte):  0 out of 18  Avg Time: /
root@ebfbe281b202:/FlashSyn#
```

The **output** we are getting from RQ1_CheeseBank.py clearly shows that the *CheeseBank benchmark is not being solved correctly* by either the polynomial (*poly*) or interpolated (*inte*) methods:

- **IDP = TDP = 2679** → no data added, suggesting that no counterexamples were found
- **Profit/Time = /** → no significant profit was found (*<= 40*) or the log file did not provide a valid time

Let us analyze the **CheeseBank benchmark log file** (*Results-To-Reproduce/FlashSynData/2000+X*) for only one of the methods used (polynomial regression or interpolation):

```
root@ebfbe281b202:/FlashSyn/Results-To-Reproduce/FlashSynData/2000+X# ls
CheeseBank_inte.txt  Eminence_inte.txt  Wdoge_poly.txt    bEarnFi_poly.txt  bZx1_poly.txt
CheeseBank_poly.txt  Wdoge_inte.txt     bEarnFi_inte.txt  bZx1_inte.txt
root@ebfbe281b202:/FlashSyn/Results-To-Reproduce/FlashSynData/2000+X#
```

First method → *cat Results-To-Reproduce/FlashSynData/2000+X/CheeseBank_inte.txt*

We analyze different sections of the log file obtained as output:

```
 ===== Strength:  0  Last Profit:  0   =====
Now global best profit is,  0.6000000000000001
For Symbolic Attack Vector:  SwapUniswapETH2LP, LP2LQ, SwapUniswapETH2Cheese, SwapUniswapCheese2ETH, RefreshCheeseBank,
BorrowCheese_USDC, BorrowCheese_USDT, BorrowCheese_DAI
        [1, 1, 1, 3322, 2215, 1326, 94]
       Estimated Profit -890.9266567456768      Actual Profit 0
 ===== Best Profit:  0.0  Best Paras:  [] ,   time:  1591.2785892486572
 ===== Strength:  0  Last Profit:  0   =====
Now global best profit is,  0.6000000000000001
```

- The system is **iterating** correctly on *symbolic attack vectors*
- Although symbolic parameters are tried, *no attack produces a concrete profit* (**Actual Profit = 0**)
- A **symbolic "Best Global Profit" of 0.6** is recorded, but it is **not concretely validated**, so it does not count in the results of the script

Note: this type of output is repeated for many lines.

```
Now global best profit is,  0.6000000000000001
===== in total  2278  concrete attack vectors are checked ======
==== in total  0  executions succeed
===== Next round we have  775  symbolic attack vectors to check:
===== Pruning 1 choose the best  100 trace candidates
===== Next round we have  100  symbolic attack vectors to check:
 ================================================================================
 =========== Strength 0 - round 0 of concrete attack vector verification finishes ================
 =========== Best Global Profit: 0.6000000000000001 ================================
 ================================================================================
SwapUniswapETH2LP number of points:
 skip
SwapUniswapETH2Cheese number of points:
 skip
RefreshCheeseBank number of points:
2000
LP2LQ number of points:
567
BorrowCheese_USDC number of points:
 skip
BorrowCheese_USDT number of points:
 skip
BorrowCheese_DAI number of points:
112
SwapUniswapCheese2ETH number of points:
 skip
```

- None of the attacks were successful, although as many as **2278 concrete carriers** were tested
- The log correctly states the **"Best Global Profit" as 0.600**…
- However, the profit was not concretely validated (**0 executions succeed**), so even though the symbolic yields an estimated profit, it is not actually realized in test foundry
- Points were generated for some functions (*RefreshCheeseBank, LP2LQ, BorrowCheese_DAI*), but many others are "*skipped*," suggesting that no data were generated for these or were ignored due to lack of coverage

```
Check Contract:         SwapUniswapETH2Cheese, SwapUniswapCheese2ETH    Profit of Previous Interation:  0.60000000000000
01  time: 1591.3402297496796
The optimizer takes 0.013136148452758789 seconds
best para: [1.e+00 3.e+05] best profit: 18956.58813860811
Optimization terminated successfully.    Next only show the first 5/5 profitable solutions
[1, 1]    estimated profit is, -443.0683038601477
[1, 24172]        estimated profit is, 2476.8626445061295
[1, 300000]       estimated profit is, 18956.58813860811
[7, 295057]       estimated profit is, 18130.34057407173
[657, 159375]     estimated profit is, -19635.193317938396

Check Contract:         RefreshCheeseBank, SwapUniswapETH2Cheese, RefreshCheeseBank, SwapUniswapETH2Cheese, SwapUniswapC
heese2ETH    Profit of Previous Interation:     0.4  time: 1591.377363204956
The optimizer takes 0.05618739128112793 seconds
best para: [1.e+00 1.e+00 3.e+05] best profit: 18850.0278246223
Optimization terminated successfully.    Next only show the first 2/2 profitable solutions
[1, 1, 300000]    estimated profit is, 18850.0278246223
[2, 3, 292456]    estimated profit is, 18162.06157726071
[1, 1, 299400]    estimated profit is, 18830.739722751696
[1, 2, 291871]    estimated profit is, 18477.00188531007
```

These blocks show **estimated profits** (e.g., 18,956 and 18,850), with relative optimal solutions found by the optimizer. But:

- No subsequent line is found that reports an Actual Profit other than zero (i.e., a concretely successful attack)
- These **profits** are only "*estimated*" and are not confirmed by testing with foundry

Although there are *estimated profit > 0*, the absence of *Actual Profit > 0* fully justifies the result reported by the script:

- **Profit**: 0
- **Time**: '/'
- **Solved**: No

```
forge test --match-contract attackTester --fork-url https://rpc.ankr.com/eth/d81f3fbb1f894af172b06e04687b43b7d94d335c233
1656722ede40d9888a46e --fork-block-number 11205646
b'Compiling 2 files with 0.7.6\nSolc 0.7.6 finished in 5.77s\nCompiler run successful (with warnings)\nwarning[5574]: sr
c/attack.sol:18:1: Warning: Co
Running attacks on foundry costs time:  16.143885374069214  seconds
For Symbolic Attack Vector:  SwapUniswapETH2Cheese, SwapUniswapCheese2ETH
        [1, 1]
       Estimated Profit -443.0683038601477      Actual Profit 0
        [1, 24172]
       Estimated Profit 2476.8626445061295      Actual Profit 0
        [1, 300000]
       Estimated Profit 18956.58813860811       Actual Profit 0
        [7, 295057]
       Estimated Profit 18130.34057407173       Actual Profit 0
        [657, 159375]
       Estimated Profit -19635.193317938396     Actual Profit 0
        [1, 24123]
       Estimated Profit 2489.2925499547914      Actual Profit 0
        [1, 299400]
       Estimated Profit 18937.517269459902      Actual Profit 0
        [6, 294466]
       Estimated Profit 18248.61381042906       Actual Profit 0
 ===== Best Profit:  0.6000000000000001  Best Paras:  [] ,   time:  1680.664190530777
 ===== Strength:  0  Last Profit:  0.6000000000000001   =====
Now global best profit is,  0.6000000000000001
```

Note: none of the attacks generated produce a profit when executed with foundry.

```
forge test --match-contract attackTester --fork-url https://rpc.ankr.com/eth/d81f3fbb1f894af172b06e04687b43b7d94d335c233
1656722ede40d9888a46e --fork-block-number 11205646
b'Compiling 2 files with 0.7.6\nSolc 0.7.6 finished in 302.12ms\nCompiler run successful\n'
Running foundry costs time:  6.022376298904419  seconds
 ================================================================================
 ================ in total  0  number of new data points added ===========
 ================================================================================
 ================================================================================
 ================================================================================
 =========== round  1  of counter-example driven loop finishes ================
 ================================================================================
 ================================================================================
SwapUniswapETH2LP number of points:
 skip
SwapUniswapETH2Cheese number of points:
 skip
RefreshCheeseBank number of points:
2000
LP2LQ number of points:
567
BorrowCheese_USDC number of points:
 skip
BorrowCheese_USDT number of points:
 skip
BorrowCheese_DAI number of points:
112
SwapUniswapCheese2ETH number of points:
 skip
```

This section of the log reports the **closing of the first synthesis loop with counterexamples** (counterexample driven loop) for CheeseBank-inte.

```
Check Contract:         SwapUniswapETH2Cheese, SwapUniswapCheese2ETH    Profit of Previous Interation:  0.60000000000000
01  time: 1686.8664381504059
The optimizer takes 0.13105511665344238 seconds
best para: [1.e+00 3.e+05] best profit: 18956.58813860811
Optimization terminated successfully.    Next only show the first 5/25 profitable solutions
[1, 39936]        estimated profit is, 4122.48138026119
[1, 300000]       estimated profit is, 18956.58813860811
[3, 294550]       estimated profit is, 18511.045964504854
[5, 292895]       estimated profit is, 18237.839315737034
[15, 295464]      estimated profit is, 17333.268040979845

Check Contract:         RefreshCheeseBank, SwapUniswapETH2Cheese, RefreshCheeseBank, SwapUniswapETH2Cheese, SwapUniswapC
heese2ETH    Profit of Previous Interation:     0.4  time: 1687.0371353626251
The optimizer takes 1.2179107666015625 seconds
best para: [1.e+00 1.e+00 3.e+05] best profit: 18850.0278246223
Optimization terminated successfully.    Next only show the first 5/20 profitable solutions
[1, 1, 1]         estimated profit is, -886.2652280048642
[1, 1, 69341]     estimated profit is, 6448.798217956855
[1, 1, 300000]    estimated profit is, 18850.0278246223
[1, 2, 297361]    estimated profit is, 18581.21375441146
[3, 5, 292573]    estimated profit is, 17912.100667367427
```

```
forge test --match-contract attackTester --fork-url https://rpc.ankr.com/eth/d81f3fbb1f894af172b06e04687b43b7d94d335c233
1656722ede40d9888a46e --fork-block-number 11205646
b'Compiling 2 files with 0.7.6\nSolc 0.7.6 finished in 5.40s\nCompiler run successful (with warnings)\nwarning[5574]: sr
c/attack.sol:18:1: Warning: Co
Running attacks on foundry costs time:  8.232158184051514  seconds
For Symbolic Attack Vector:  SwapUniswapETH2Cheese, SwapUniswapCheese2ETH
        [1, 39936]
       Estimated Profit 4122.48138026119        Actual Profit 0
        [1, 300000]
       Estimated Profit 18956.58813860811       Actual Profit 0
        [3, 294550]
       Estimated Profit 18511.045964504854      Actual Profit 0
        [5, 292895]
       Estimated Profit 18237.839315737034      Actual Profit 0
        [15, 295464]
       Estimated Profit 17333.268040979845      Actual Profit 0
        [139, 232845]
       Estimated Profit 5547.27699248945        Actual Profit 0
        [171, 278339]
       Estimated Profit 8229.580789661022       Actual Profit 0
        [247, 99610]
```

Note: same structure as the output obtained previously.

```
For Symbolic Attack Vector:  RefreshCheeseBank, SwapUniswapETH2LP, LP2LQ, BorrowCheese_USDC, SwapUniswapETH2Cheese, Borr
owCheese_USDT, SwapUniswapCheese2ETH, BorrowCheese_DAI
        [1, 33436, 151484, 11095, 414680, 44825, 10179]
       Estimated Profit -55883.567270044005     Actual Profit 0
 ===== Best Profit:  0.0  Best Paras:  [] ,   time:  7822.147076129913
 ===== Strength:  1  Last Profit:  0.2   =====
Now global best profit is,  1.6
===== in total  429  concrete attack vectors are checked ======
==== in total  0  executions succeed
===== Next round we have  100  symbolic attack vectors to check:
 ================================================================================
 =========== round  2  of concrete attack vector verification finishes ==========
 =========== Best Global Profit: 1.6 ===============================
 ================================================================================
forge test --match-contract attackTester --fork-url https://rpc.ankr.com/eth/d81f3fbb1f894af172b06e04687b43b7d94d335c233
1656722ede40d9888a46e --fork-block-number 11205646
b'Compiling 2 files with 0.7.6\nSolc 0.7.6 finished in 307.51ms\nCompiler run successful\n'
Running foundry costs time:  2.356527805328369  seconds
 ================================================================================
 ================ in total  0  number of new data points added ===========
 ================================================================================
 ================================================================================
 ================================================================================
 =========== round  2  of counter-example driven loop finishes ================
 ================================================================================
 ================================================================================
```

Let us analyze what happens in **Round 2 of the inte method for the CheeseBank benchmark**:

- Even if the profit had not been concretely made (all *Actual Profit = 0*), FlashSyn found a higher estimated potential profit in round 2 (from 0.6 to 1.6), so it *updates the "best global profit*
- The system is working correctly, but fails to produce verifiable profitable attacks

```
Check Contract:         RefreshCheeseBank, SwapUniswapETH2LP, LP2LQ, SwapUniswapETH2Cheese, BorrowCheese_DAI, BorrowChee
se_USDC, SwapUniswapCheese2ETH, BorrowCheese_USDT    Profit of Previous Interation:     0.4  time: 8145.3838312625885
The optimizer takes 58.74723696708679 seconds
best para: [1.00000000e+00 5.75523210e+01 1.00000000e+00 8.75860000e+04
 3.68690175e+05 3.00000000e+05 2.12881688e+04] best profit: 474793.0119027301
Optimization terminated successfully.    Next only show the first 2/2 profitable solutions
[1, 57, 1, 87586, 368690, 300000, 21288]          estimated profit is, 474793.0119027301
[29, 901, 15432, 21255, 178751, 270263, 180743]           estimated profit is, 344408.01487132703
[1, 56, 1, 87410, 367952, 299400, 21245]          estimated profit is, 473859.32040719455
[28, 899, 15401, 21212, 178393, 269722, 180381]           estimated profit is, 345971.4179935864

Check Contract:         RefreshCheeseBank, SwapUniswapETH2LP, LP2LQ, SwapUniswapETH2Cheese, BorrowCheese_DAI, BorrowChee
se_USDC, BorrowCheese_USDT, SwapUniswapCheese2ETH    Profit of Previous Interation:     0.4  time: 8204.37356185913
The optimizer takes 60.078553199768066 seconds
best para: [1.00000000e+00 7.24063891e+01 1.00000000e+00 8.75860000e+04
 3.20859278e+05 5.15996821e+04 3.00000000e+05] best profit: 426962.1145129993
Optimization terminated successfully.    Next only show the first 2/2 profitable solutions
[1, 72, 1, 87586, 320859, 51599, 300000]          estimated profit is, 426962.1145129993
[87, 509, 15863, 83779, 148454, 122712, 186474]           estimated profit is, 157221.75618578758
[1, 71, 1, 87410, 320217, 51495, 299400]          estimated profit is, 477619.32040719455
[86, 507, 15831, 83611, 148157, 122466, 186101]           estimated profit is, 159462.1302208551

root@ebfbe281b202:/FlashSyn#
```

In the **last rows** there're are:

- **Complex Symbolic Attack Vectors**, combining multiple features (e.g., BorrowCheese_USDC, BorrowCheese_USDT, etc.).
- **Estimated profits are very high** (up to ~476k) but again, no rows with Actual Profit are reported, so it is likely that Actual Profit = 0 again as before.

Note: the results obtained are consistent with what the pyhton script gives us but not with the results in the paper (table 3).

Subsection 3.2: RQ2

RQ2 says: "*How well does the synthesis-via-approximation technique perform compared to precise baselines?*".

bEarnFi and Wdoge: Execution

We always work on the two benchmarks used to answer RQ1: **bEarnFi** and **Wdoge**. First, we go to run the **bash script runRQ2.sh** → *chmod +x ./runRQ2.sh*

./runRQ2.sh bEarnFi

./runRQ2.sh Wdoge

```
root@ec0ef2b40145:/FlashSyn# chmod +x ./runRQ2.sh
root@ec0ef2b40145:/FlashSyn# ./runRQ2.sh bEarnFi
========= Current Date and Time: Thu May  1 16:34:45 UTC 2025 =========
running FlashSyn-precise baseline for bEarnFi...
bEarnFi (precise) done.
Benchmarks: 1/7 finished, 6/7 to do
================================================================================================================
========================================== ALL COMPLETE ==========================================
================================================================================================================
root@ec0ef2b40145:/FlashSyn# ./runRQ2.sh Wdoge
========= Current Date and Time: Thu May  1 16:36:29 UTC 2025 =========
running FlashSyn-precise baseline for Wdoge...
Wdoge (precise) done.
Benchmarks: 1/7 finished, 6/7 to do
================================================================================================================
========================================== ALL COMPLETE ==========================================
================================================================================================================
root@ec0ef2b40145:/FlashSyn#
```

Since our analysis will be oriented to only two benchmarks, it is necessary, as in the previous case, to **modify the python script (RQ2.py)**:

```python
def getProfitinHistory(benchmark: str):
    profit = 0
    # TEST 1
    if benchmark == "bEarnFi":
        profit = 18077.148053847253
    elif benchmark == "Wdoge":
        profit = 78

    return int(profit)


def main():
    method = 0 # 0 for interpolation
               # 1 for polynomial

    # TEST 1
    benchmarkList = ['bEarnFi', 'Wdoge']
```

Then, move it to the container:

```
lm@lucam:~/project_FlashSyn/script_test$ docker cp RQ2_test1.py ec0ef2b40145:/FlashSyn/Results-To-Reproduce
Successfully copied 4.1kB to ec0ef2b40145:/FlashSyn/Results-To-Reproduce
```

We launch the modified analysis script → *python3 Results-To-Reproduce/RQ2_test1.py*

```
root@ec0ef2b40145:/FlashSyn# python3 Results-To-Reproduce/RQ2_test1.py
benchmarkprecise
bEarnFi /
Wdoge /
root@ec0ef2b40145:/FlashSyn#
```

bEarnFi and Wdoge: Analysis of Results

Now, we analyze the files in the **FlashSyn/Results-To-Reproduce/FlashSynData/precise folder**:

```
root@ec0ef2b40145:/FlashSyn/Results-To-Reproduce/FlashSynData/precise# ls
Wdoge_precise.txt  bEarnFi_precise.txt
```

For **bEarnFi**:

```
root@ec0ef2b40145:/FlashSyn/Results-To-Reproduce/FlashSynData/precise# cat bEarnFi_precise.txt
```

Results:

```
Deposit number of points:
 skip
EmergencyWithdraw number of points:
0
 ================================================================================
 ======================== End of Synthesis, time in total:  57.75315546989441 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
 ======================== End of Answers    ====================================
 =================== Best Profit  19.200000000000003  ============================
```

For **Wdoge**:

```
root@ec0ef2b40145:/FlashSyn/Results-To-Reproduce/FlashSynData/precise# cat Wdoge_precise.txt
```

Results:

```
SwapWBNB2Wdoge number of points:
 skip
TransferWdoge number of points:
 skip
PancakePairSkim number of points:
 skip
PancakePairSync2 number of points:
 skip
SwapWdoge2WBNB number of points:
 skip
 ================================================================================
 ======================== End of Synthesis, time in total:  72.46661829948425 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
 ======================== End of Answers    ====================================
 ==================== Best Profit  4.2  ==============================
```

Note: the files bEarnFi_precise.txt and Wdoge_precise.txt exist and contain data, but the Python script returns you "**/**", the problem is in the contents of the file: most likely it does not contain the **string "Best Global Profit:"**, which is the one the script looks for to extract the profit.

Let's see if the line indicating the maximum profit exists with **grep**:

- *grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/bEarnFi_precise.txt*

```
root@ec0ef2b40145:/FlashSyn# grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/bEarnFi_precise.txt
 =========== Best Global Profit: 0.2 ================================
 =========== Best Global Profit: 0.2 ================================
 =========== Best Global Profit: 5.0 ================================
 =========== Best Global Profit: 5.0 ================================
 =========== Best Global Profit: 19.200000000000003 ================================
 =========== Best Global Profit: 19.200000000000003 ================================
root@ec0ef2b40145:/FlashSyn#
```

- *grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/Wdoge_precise.txt*

```
root@ec0ef2b40145:/FlashSyn# grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/Wdoge_precise.txt
 =========== Best Global Profit: 0.4 ================================
 =========== Best Global Profit: 0.4 ================================
 =========== Best Global Profit: 1.2000000000000002 =================================
 =========== Best Global Profit: 1.2000000000000002 =================================
 =========== Best Global Profit: 4.2 ================================
 =========== Best Global Profit: 4.2 ================================
root@ec0ef2b40145:/FlashSyn#
```

Analyzing the python script, I realized that the printing of the value of the maximum profit is done if 30 (**threshold**) is exceeded. Since in both cases the threshold is not exceeded, then "**/**" is printed.

```
75          if globalbestProfit > 30:
76              print(int(globalbestProfit), end = " ")
77          else:
78              print("/", end = " ")
```

bEarnFi and Wdoge: Comparison

Let us now compare with the values in the folder of results already obtained (those in the paper):

```
root@ec0ef2b40145:/FlashSyn/Results-Expected/FlashSynData/precise# ls
CheeseBank_precise.txt  PuppetV2_precise.txt  Puppet_precise2.txt  Wdoge_precise.txt
Eminence_precise.txt    Puppet_precise.txt    Warp_precise.txt     bEarnFi_precise.txt
```

For bEarnFi:

```
root@ec0ef2b40145:/FlashSyn/Results-Expected/FlashSynData/precise# cat bEarnFi_precise.txt
```

Results:

```
Deposit number of points:
 skip
EmergencyWithdraw number of points:
0
 ================================================================================
 ======================== End of Synthesis, time in total:  110.64523196220398 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
For Symbolic Attack Vector:  Deposit, EmergencyWithdraw, Deposit, EmergencyWithdraw
Best Profit:  13832   Parameters:  [7804238, 7800912]
 ======================== End of Answers    ==================================
 ==================== Best Profit  13832  ============================
root@ec0ef2b40145:/FlashSyn/Results-Expected/FlashSynData/precise#
```

For Wdoge:

```
root@ec0ef2b40145:/FlashSyn/Results-Expected/FlashSynData/precise# cat Wdoge_precise.txt
```

Results:

```
SwapWBNB2Wdoge number of points:
 skip
TransferWdoge number of points:
 skip
PancakePairSkim number of points:
 skip
PancakePairSync2 number of points:
 skip
SwapWdoge2WBNB number of points:
 skip
 ================================================================================
 ======================== End of Synthesis, time in total:  86.553226947784442 s ==============
 ================================================================================
 ======================== Now shows the answers: ========================================
For Symbolic Attack Vector:  SwapWBNB2Wdoge, TransferWdoge, PancakePairSkim, PancakePairSync2, SwapWdoge2WBNB
Best Profit:  75   Parameters:  [2859, 5156250, 4609375]
 ======================== End of Answers    ====================================
 =================== Best Profit  75  ============================
root@ec0ef2b40145:/FlashSyn/Results-Expected/FlashSynData/precise#
```

Comparison for bEarnFi:

Aspect	Reproduce Results	Expected Results	Difference
Synthesis Time	~57.75 s	~110.64 s	First is faster (less exploration?)
Valid Data Points	skip / 0	skip / 0	-
Attack Vector	Not shown	Deposit, EmergencyWithdraw, Deposit, EmergencyWithdraw	First output is truncated or simplified
Attack Parameters	Not shown	[78004238, 7800912]	Not available in first output
Best Profit	19.2	13832	Much lower in first case

The first execution found a **much less profitable solution**. This is expected, as written in the README and paper: *"...results may not be exactly the same...the solver shgo adopts different strategies based on hardware."* Also, the solver may:

- Stop at a **local optimum** (e.g., profit 19.2) without exploring other regions.
- Be limited by timeout or availability of **cores/processors** (the paper used up to 18).

Comparison for Wdoge:

Aspect	Reproduce Results	Expected Results	Difference
Synthesis Time	~72.47 s	~86.55 s	Similar, so time is not the issue
Valid Data Points	skip / skip	skip / skip	Equal
Attack Vector	Not shown	SwapWBNB2Wdoge, TransferWdoge, PancakePairSkim, PancakePairSync2, SwapWdoge2WBNB	First output is truncated or simplified
Attack Parameters	Not shown	[2859, 5156250, 4609375]	Not available in first output
Best Profit	4.2	75	Much lower in first case

Even in this case, the first execution found a **much less profitable solution**. The solver may:

- The local run does **not collect data** for key functions, or the solver does not converge on an effective attack vector.
- You may have had **hardware/software variations** (e.g., optimizer, number of processes, timing limit), or different versions of the initial data.

CheeseBank: Execution

First, we go to run the **bash script runRQ2.sh** → *chmod +x ./runRQ2.sh*

./runRQ2.sh CheeseBank

```
root@ebfbe281b202:/FlashSyn# chmod +x ./runRQ2.sh
root@ebfbe281b202:/FlashSyn# ./runRQ2.sh CheeseBank
========= Current Date and Time: Mon May  5 08:31:56 UTC 2025 =========
running FlashSyn-precise baseline for CheeseBank...
CheeseBank (precise) done.
Benchmarks: 1/7 finished, 6/7 to do
================================================================================================================
========================================== ALL COMPLETE ==========================================
================================================================================================================
root@ebfbe281b202:/FlashSyn#
```

Since our analysis will be oriented to a specific benchmark, it is necessary, as in the previous case, to **modify the python script (RQ2.py)**:

```python
def getProfitinHistory(benchmark: str):
    profit = 0
    if benchmark == "CheeseBank": # TEST
        profit = 3270347.8

    return int(profit)


def main():
    method = 0 # 0 for interpolation
               # 1 for polynomial

    benchmarkList = ['CheeseBank'] # TEST
```

Then, move it to the container:

```
lm@lucam:~/project_FlashSyn/script_test$ docker cp RQ2_CheeseBank.py ebfbe281b202:/FlashSyn/Results-To-Reproduce
Successfully copied 4.1kB to ebfbe281b202:/FlashSyn/Results-To-Reproduce
lm@lucam:~/project_FlashSyn/script_test$
```

We launch the modified analysis script → *python3 Results-To-Reproduce/RQ2_CheeseBank.py*

```
root@ebfbe281b202:/FlashSyn# python3 Results-To-Reproduce/RQ2_CheeseBank.py
benchmarkprecise
CheeseBank New global Profit:  0.6000000000000001
/
root@ebfbe281b202:/FlashSyn#
```

- **New global Profit: 0.600...01** → the script found at least a token profit, but not high enough (below the useful threshold, e.g., 40) to be recorded as an actual exploit
- The **final /** represents no Actual Profit > 0 or no valid runtime

CheeseBank: Analysis of Results

Now, we analyze the files in the **FlashSyn/Results-To-Reproduce/FlashSynData/precise folder**:

```
root@ebfbe281b202:/FlashSyn# cd Results-To-Reproduce/FlashSynData/precise/
root@ebfbe281b202:/FlashSyn/Results-To-Reproduce/FlashSynData/precise# ls
CheeseBank_precise.txt  Wdoge_precise.txt  bEarnFi_precise.txt
```

For **CheeseBank**:

```
root@ebfbe281b202:/FlashSyn/Results-To-Reproduce/FlashSynData/precise# cat CheeseBank_precise.txt
```

Results:

```
Check Contract:         RefreshCheeseBank, SwapUniswapETH2LP, SwapUniswapETH2Cheese, LP2LQ, BorrowCheese_DAI, BorrowChee
se_USDC, BorrowCheese_USDT, SwapUniswapCheese2ETH    Profit of Previous Interation:     0.2  time: 7621.208479166031
The optimizer takes 112.23292708396912 seconds
best para: [1.00000000e+00 1.00000000e+00 7.84336677e+00 8.75860000e+04
 8.09548591e+02 2.16798619e+02 3.00000000e+05] best profit: 18516.836811552836
Optimization terminated successfully.    Next only show the first 1/1 profitable solutions
[1, 1, 7, 87586, 809, 216, 300000]        estimated profit is, 18516.836811552836
[1, 1, 6, 87410, 807, 215, 299400]        estimated profit is, 18497.320407194526

Check Contract:         SwapUniswapETH2LP, LP2LQ, BorrowCheese_DAI, BorrowCheese_USDT, RefreshCheeseBank, SwapUniswapETH
2Cheese, SwapUniswapCheese2ETH, BorrowCheese_USDC    Profit of Previous Interation:     0.2  time: 7735.72798538208
The optimizer takes 108.75004386901855 seconds
best para: [3.33256650e+01 5.72456088e+01 1.00000000e+00 1.00000000e+00
 7.32739584e+02 2.21050653e+05 1.00000000e+00] best profit: -40212.90605325316
Optimization terminated successfully.    Next only show the first 1/1 profitable solutions
[33, 57, 1, 1, 732, 221050, 1]    estimated profit is, -40212.90605325316
```

Note: despite the complexity of the analysis (numerous paths and solutions explored), none were successful, so the benchmark is “unresolved.” In addition, the structure of the output is very different from that of “simpler” benchmarks.

Let's see if the line indicating the maximum profit exists with **grep**:

- *grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/CheeseBank_precise.txt*

```
root@ebfbe281b202:/FlashSyn# grep "Best Global Profit:" Results-To-Reproduce/FlashSynData/precise/CheeseBank_precise.txt
 =========== Best Global Profit: 0.6000000000000001 ================================
 =========== Best Global Profit: 0.6000000000000001 ================================
root@ebfbe281b202:/FlashSyn#
```

CheeseBank: Comparison

Let us now compare with the values in the folder of results already obtained (those in the paper):

```
root@ebfbe281b202:/FlashSyn/Results-Expected/FlashSynData/precise# ls
CheeseBank_precise.txt  PuppetV2_precise.txt  Puppet_precise2.txt  Wdoge_precise.txt
Eminence_precise.txt    Puppet_precise.txt    Warp_precise.txt     bEarnFi_precise.txt
```

For CheeseBank:

```
root@ebfbe281b202:/FlashSyn/Results-Expected/FlashSynData/precise# cat CheeseBank_precise.txt
```

Results:

```
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_USDC, SwapUn
iswapCheese2ETH, BorrowCheese_DAI
Best Profit:  1788587.8   Parameters:  [73, 15281, 3338, 1997812, 121033, 54923]
For Symbolic Attack Vector:  SwapUniswapETH2Cheese, RefreshCheeseBank, SwapUniswapCheese2ETH, SwapUniswapETH2LP, LP2LQ, Bo
rrowCheese_USDT, BorrowCheese_DAI
Best Profit:  1144015.4   Parameters:  [8995, 135535, 84, 2007, 1201272, 77033]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_USDC, Borrow
Cheese_DAI, SwapUniswapCheese2ETH, BorrowCheese_USDT
Best Profit:  2027910.8   Parameters:  [67, 16620, 2827, 1692876, 21345, 259790, 409421]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, SwapUniswapCheese2ETH, Bo
rrowCheese_USDT, BorrowCheese_DAI, BorrowCheese_USDC
Best Profit:  1968274.8   Parameters:  [89, 16399, 3974, 198091, 1193927, 39204, 923947]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_USDT, SwapUn
iswapCheese2ETH, BorrowCheese_DAI, BorrowCheese_USDC
Best Profit:  1130250.6   Parameters:  [64, 13753, 2500, 424709, 102918, 13658, 967554]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_USDT, SwapUn
iswapCheese2ETH, BorrowCheese_USDC
Best Profit:  2043708.4   Parameters:  [74, 18430, 3086, 986363, 47805, 1756715]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_USDT, Borrow
Cheese_USDC, BorrowCheese_DAI, SwapUniswapCheese2ETH
Best Profit:  2404532.6   Parameters:  [99, 18221, 4033, 770724, 1986451, 33828, 115576]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, SwapUniswapCheese2ETH, Bo
rrowCheese_USDT, BorrowCheese_DAI
Best Profit:  1072649.0   Parameters:  [33, 18782, 1263, 198651, 1054789, 86556]
For Symbolic Attack Vector:  SwapUniswapETH2LP, SwapUniswapETH2Cheese, RefreshCheeseBank, LP2LQ, BorrowCheese_DAI, BorrowC
heese_USDT, SwapUniswapCheese2ETH, BorrowCheese_USDC
Best Profit:  2816762.4   Parameters:  [83, 15091, 4084, 23587, 1005825, 277225, 1901696]
 ====================== End of Answers    ==================================
 ================== Best Profit  2816762.4  ============================
root@ebfbe281b202:/FlashSyn/Results-Expected/FlashSynData/precise#
```

Comparison for CheeseBank:

Aspect	Reproduce Results	Expected Results	Difference
Synthesis Time	~1680–1730 s	Not explicit (but successfully completed)	Relatively long time, but still complete
Valid Data Points	Some skip - e.g., BorrowCheese_USDC, USDT	Numerous valid points shown for each contract	Reproduce seems less exploratory or more conservative
Attack Vector	Shown in part (but with 0 profit results)	Explicit with details: SwapX, Refresh, Borrow...	More complete and detailed in expected results
Attack Parameters	Often negative or profit 0	Complete parameters, with positive profits	Reproductive identifies suboptimal strategies or ignores them
Best Profit	0.6000000000000001	2816762.4	Much lower in first case, probably for a failure during the execution

The **reproduced result** potentially identified strategies, but all attempts resulted in *0 profit except* for an insignificant trace.

The synthesis worked but probably:

- It had difficulty executing the tests correctly on Foundry
- or, it generated combinations that were valid in theory but not executable in practice.

The **expected results**, in contrast, show numerous parameters explored with profits well over a million, so clearly something is not being faithfully reproduced.

Section 4: Conclusion

Main Aspects

Baseline execution: FlashSyn was successfully installed and tested via Docker, in WSL environment on Windows 11.

Overall effectiveness (RQ1): FlashSyn was able to *find attacks with positive profit* for some benchmarks (*bEarnFi, Wdoge*), but not for others (*CheeseBank*).

Approximate techniques: *Polynomial interpolation* and *regression techniques* work, but they do not always find valid or concrete attacks.

Precise technique (RQ2): In *precise tests*, estimated profits are often high but are not concretely validated (*Actual Profit = 0*).

CheeseBank: high profit estimates but no valid execution (*no concrete attacks produce profit*). Profits are often much lower than the paper's table (e.g., 0.6 vs. 2.8 million for CheeseBank).

Difficulties

Hardware and performance: *docker* uses *1 process*; paper used *18* → results slower and less exploration.

Optimizer Strategies: *shgo* is non-deterministic.

Absence of concrete attacks: Even with high estimated profits, no valid Actual Profit → causes failure in Foundry phase.

Inconsistent Output: Log files turn out structurally different than expected, especially for CheeseBank.

Fragile Python Parsing: Scripts fail if they find '**/**' characters or strings in numeric lists → modified script.

Possible errors

Differences in hardware/software (Docker, WSL, RAM, threading): my machine has limitations in comparison to the paper's multi-core configuration.

Foundry execution failure: contracts are not likely to execute correctly (e.g., runtime errors in tests).

Problems with initial data generation: some functions are skipped ("skip") → less coverage → less chance of discovering attacks.

Updates/breaks in the toolchain: possible mismatch between versions of forge, solc, or foundry, causing inconsistencies.