Viability of detecting desyncronizing disassembly using Isolation Forest

1st Edvard Thörnros
Linköping University
Linköping, Sweden
edvth289@student.liu.se

***Abstract*—This study proposes and tests a binary classifier for opaque predicates based on *Isolation Forest*. The proposed classifier did not work well.**

Opaque predicates are branch conditions which force a specific branch when running the code. The branch that is never taken can jump to an address between "valid" instructions. These branches confuses disassemblers and makes revers engineering of malware and programs harder.

Machine code distributed in an Ubuntu installation ISO are used to train an *Isolation Forest* machine learning models to recognize "normal" code after a branch instruction. The model can then say which branch is more unusual. 3 medium sized programs were compiled using "desync-cc" (a compiler that inserts opaque predicates). Two different kinds of opaque predicates were tested. The resulting machine code was used to verify the model as a binary classifier.

The binary classifier distinguished some what successfully uniform noise from machine code when looking at at most 1 x86 instruction. When more input was feed to the binary classifier the models worked less well. The models quickly degraded in quality when looking at multiple x86 instructions and failed to produce accurate predictions.

Reasoning and thoughts about why *Isolation Forest* only handles a fixed complexity are documented. The leading hypothesis is that *Isolation Forest* doesn't have enough computational power to fully understand a language as complex as x86 assembly.

I. Introduction and background

Reading and understanding program code can aid a lot in documenting, understanding, preventing and limiting malicious programs. There is always an ongoing arms race between security researchers and malware writers. Malicious program writers seek to gain an edge in any way they can, and one tool is code obfuscation.

Code obfuscation is a technique for making it hard to interpret the true meaning of a program without running it – at which point it can be too late. One specific kind of code obfuscation is dissassembly desynchronization. Disassembly desynchronization is a way to hide the intent of assembly code by fooling the disassembler. Usually a branch instruction or a jump is used that lands in an unreachable region of the program. One common way of doing disassembly desynchronization is to inject garbage data in paths the program never can take.

A. Introduction to disassembly desynchronization

Disassembly desynchronization is a collection of techniques that try to confuse the disassembler – a disassembler is a program that converts binary machine code into "high level" assembly code. For disassembly desynchronization to be possible there are a few requirements. The machine code instruction set needs to support variable length instructions – for example some instructions are 2 bytes long while other are 4 bytes long. x86-64 (also called: 64bit x86, x64, AMD64, Intel 64) is a variable length instruction set [15] and one of the most common instruction sets (as of 2022).

It is often helpful to look at an example. Consider this small extract of assembly code:

```
0x77b18: xor r11, r11
0x77b1b: je 0x77b21
0x77b1d: imul ebx,
    dword ptr [rdx + 0x2ae83fec], 0x11
0x77b24: sbb eax, dword ptr [rax]
```

There are four lines in this example but we only need to understand the first two. Let's break this example down.

The first instruction: `xor r11, r11`, does an xor on the register `r11` with itself – this is equivalent to setting the register to 0 and updating flags. Specifically it sets the flag `ZF=1` (zero flag).

The second instruction: `je 0x77b21`, jumps to the address `0x77b21` if `ZF` is set, which it always will be. We can think of these instructions combined as a guaranteed jump to `0x77b21`. [5]

This also means that the disassembled code seen above is irrelevant after the `je` instruction. The instructions at offset `0x77b1d` and `0x77b24` will never be run. Note that the offset `0x77b21` lies in between the disassembled instructions. In this example, junk bytes have been inserted after the branch and before address `0x77b21`. This causes a disassembly desynchronization. The junk bytes attempt to confuse the disassembler. If we wanted to see what instructions would run if the branch is taken – which we might not know when analyzing this program – we would have to disassemble the program again starting at `0x77b21`. Here's the rub. These branches are easy to make, and can require significant amount of time and effort to understand.

The short example above is a case of "opaque predicates". It is known when the program is written that the jump will always be taken – since the predicate is always true – but the fact is hidden when reading the disassembly. In this case the predicate makes it easy (trivial even) to see which branch is

taken. But code could be more complex or involve multiple processes, making it a lot harder to untangle the unreachable code. [11]

B. Methods of disassembly desynchronization in this paper

There are two methods of disassembly desynchronization referenced in this paper.

1) With junk bytes: Here junk data is inserted after a branch instruction. The junk data is has to be a valid x86 instruction. But the junk data is never run. The branch is therefore always taken. The branch lands at an instruction between instructions – if we parse the junk bytes. This is explained in more detail in section I-A.

2) Without junk bytes: Here the branch is **never** taken. The branch lands at an offset in the middle of an existing instruction. But if we parse the machine code from where the branch instruction lands, we can get other instructions.

C. Methods to detect disassembly desynchronization

We will focus more precisely on detecting invalid branches after opaque predicates – branches of code that is never taken. There are methods that look at the instructions leading up to to the branch. Either through simulating the code, sophisticated machine learning techniques or by using heuristics. [12]

Another possibility is to look at the instructions/bytes after a jump, and running a ”sanity-check” on the branches. This second approach is what this paper will focus on. More precisely if it can be done using the *isolation forest* machine learning technique. [3]

D. Isolation forest – a technique for unsupervised machine learning

Isolation forest is an algorithm that lets a computer detect anomalies in a data set. The basis for the algorithm is that is harder to describe a ”normal” piece of data than an ”unusual” piece of data. The data set is split many times using random ”split values” (which can be thought of as trees) until each sample is isolated (hence the name). The number of splits required to uniquely identify a sample is referred to as the ”path-length” for a sample. The ”path-length” can be thought of as a measure of how ”unusual” the sample is. One aspect isolation forest is said to handle better than many other anomaly detection methods is data with high dimensionality. Machine code has high dimensionality, since each byte can be thought of as a dimension. [3]

This paper will use scikit-learn’s implementation of isolation forest. [10]

E. Tools

GCC One of the most commonly used C/C++ compilers – but can also compile other languages.

Desync-CC is a drop-in replacement for GCC that can inject opaque predicates and invalid branches into a program while compiling. [6]

scikit-learn learn is a collection of implemented machine learning algorithms for the Python programming language. [9]

F. The research question and a brief overview of the study

Can the isolation forest algorithm be used to identify disassemble desynchronization?

There have been previous works summarized by Ucci **andothers**. These include successful applications of machine learning models in malware classifications. There are also methods with dataflow analysis. Though these methods have limitations of their own. [14] [8] [11] [12] But little research has been done applying isolation forest to detect disassembly desynchronization.

This paper aims to collect data, train a simple machine learning model using the data then using the model as a binary classifier on pairs of valid and invalid paths.

There are some interesting observations made about the machine code generated by commonly used compilers. Both Clang and GCC do **not** generate jump instructions landing at invalid program offsets. Clang and GCC keeps data and program code separate in their generated binaries. That is, Clang and GCC does not generate disassembly desynchronizing machine code. Since most binaries on the internet are compiled using these compilers, it should be very easy to find ”valid” code to train on. [1]

G. Expected results

Since similar works have been tried successfully with other machine learning methods, it’s expected that this technique will work and the machine model will be able to distinguish successfully between valid and invalid paths. [12] [14] [8] A model that works more than 80% of the time would be a model that works well. There is a lot of randomness and noise involved in this process. The experiments are also set up to give isolation forest optimal conditions. If the model cannot perform close to 100% here – it’s just not good enough.

Note that since we are making a binary classifier we want the models to be far from 50%. Since given a binary classifier that is always wrong. We can negate it’s output. This gives us a binary classifier that is always right.

II. Method

Data Collection. To be able to apply machine learning data needs to be collected. To this end, the internet was searched for small sized C-programs. In the end Git [4], Curl [2] and a single file version of Gzip [7] were used.

These projects were compiled with Desync-CC [6] using 2 configurations. The configurations for Desync-CC is included in the appendix, one configuration with junk (VI-A) and one configuration without junk (VI-B). These data sets will be referred to as the verification sets. The tool Desync-CC [6] inserts labels for all desync points – a desync point is is the start of the instruction after the desyncronizing jump. From these compiled libraries and executable files all desync points were listed and branches were extracted using the following methods.

For the generated data with junk-bytes present VI-A. These desync points contain information about the number of junk bytes. This information makes it easy to extract the bytes

before and after the junk bytes. The path that jumps is always the correct path, and is labeled as such in the data. Both paths are stored together in a labeled pair.

For the generated data where there's no junk bytes VI-B. The instruction right before these desync points contains a jump instruction. And the byte right before the desync point contains the jump offset. We then double check that this jump lands inside an instruction - not at the start of a parsed instruction. The path that does not jump is always the correct path, and is labeled as such in the data. Both paths are stored together in a labeled pair.

Binaries were extracted from a Linux installation ISO. A simple bash script was used to extract all X86 ELF dynamic libraries and executable file from the latest available Ubuntu LTS installation ISO (20.04.3 – latest version at time of writing) [13]. A simple python script was then used to disassembled the binaries (libraries and executable files). Up to 50 bytes after the jump instruction and the jump destination were copied out. All the data was then de-duplicated. This data set will be referred to as the training set.

All data was then stored base-64 encoded in a JSON files ready for the training step.

Training. One variable was varied - the number of bytes of look ahead for each branch. The byte look ahead was varied between 1 and 50 bytes. The training set was first trimmed so all pieces of training data is the desired number of bytes long. If some samples are too short they are filtered out. All samples are then used to train the model. Using the scikit-learn option:

```
# The 1.0 means use 100% of the
# available data to train on.
max_samples=1.0
# The number of trees.
n_estimators=1000
# The number of cores to train on.
n_jobs=12
```

All other options were left to their default values.

The training set had around 200000 unique data points.

Verification. We then run the model on the verification sets using the $decision\_function$. The $decision\_function$ is our classifier – but it gives out an "anomaly score".

Then we compare the "anomaly score" of these branches too see if one looks more "odd" than the other. If the desyncronised path is more "odd" than the correct path we label the test a success. The fraction of the number of tests that pass can then be plotted.

The verification sets had around 5000 unique data points each.

Histograms were also generated for all data sets – both the verification set and the training sets. The histograms had the first byte of all samples.

III. Limitations and Further Research

There are a lot of limitations in this study. Most of the limitations arise from time constraints since this paper was written as part of the course *TDDD17 – Information Security, Second Course* at Linköping's University.

A. Not finding the data "in the wild"

All the data was either generated or collected from assumed good binaries or generated using a tool. This might make the data-set very "unnatural" and might not at all represent how good this approach is on actual data. But the method would still describe a "best case" for the algorithm.

All machine learning algorithms have to have seen at least parts of a problem to be able to solve it successfully. It's possible there are instances of disassembly desynchronization which this model won't be able to distinguish from valid programs. This model might need to be paired with a sophisticated algorithm or human judgment to always be correct.

The histogram in Figure 2 shows that there are big differences in the statistical properties between the first bytes of our verification set with junk bytes. It would be trivial for a malware writer to generate junk bytes which follows the statistical properties of instructions after jumps. This further cements the problems of this method.

B. Possible over-fitting to compiler optimizations

There is one potential difference between the code from the Ubuntu installation medium and the code compiled as part of this project. The Ubuntu binaries might have been compiled with optimizations – to make the model more general programs compiled with and without optimizations should be used.

C. Tricking models of this kind

One very easy counter attack against a model like this is to simply inject valid looking code for the look ahead length. After the 50 bytes – since we used 50 bytes in this paper – you can let the code devolve into garbage data. This would force the model to be completely retrained since the model **must have** a finite look ahead.

D. What are the limits of what Isolation Forest can learn?

There could possibly be a mapping between computational complexity and the computational power of isolation forest. Maybe a variable length instruction set is simply too hard for a model of this complexity to learn.

E. Maybe a well tuned statistical algorithm can outperform isolation forest

The use of histograms in this paper leads to the intuition behind a very simply algorithm. An algorithm that simply calculates the odds of bytes being at each position if the program should "look valid". This algorithm could very possibly outperform the isolation forest.

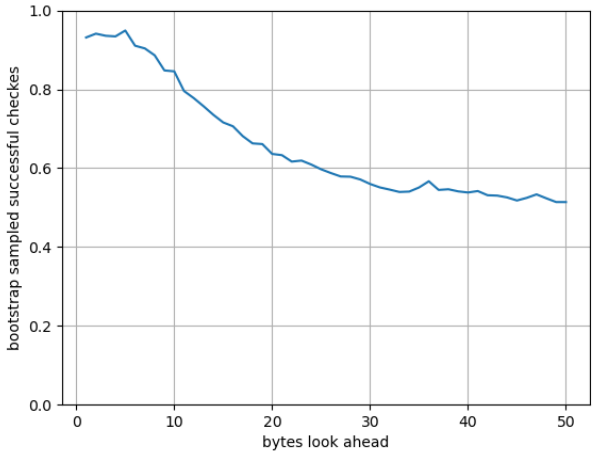

Fig. 1. Odds for path pairs to be labeled correctly. That is the invalid path was seen as ”more odd”. For short byte look ahead – 1 to 5 bytes – the model works well. For longer byte look ahead, the model performs very poorly.

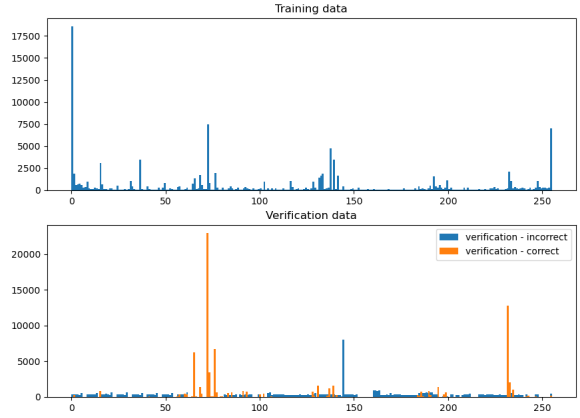

Fig. 2. Histogram over the first bytes in the training data and the verification data generated with junk.

IV. Results

A. Results with junk data

When the model was run on verification data generated by VI-A. Figure 1 shows that the model is highly accurate with short look ahead. The machine models with more data perform worse.

The Figure 1 shows that the classification gets a lot worse once we get past the first initial bytes of junk-data. Since there is a suspicious peek at the 4-byte look ahead point. This is also the size of the junk data. It is therefore possible that more data makes the model more uncertain. The models seem to have a difficult time differentiating between relevant and irrelevant information.

This specific case is a very poor representative of how well this technique would work in practice. Judging from Figure 1 the models struggle when fed more data. The verification set does not match the statistical features of the training data. Some bytes are over represented compared to ”valid” code. This dataset therefore unfairly favors the model as it is. Classification is trivial for anyone with access to the histogram of the training data.

The histogram in Figure 2 shows that the first byte of the invalid path is very likely to differ from the training data and actual programs. First bytes of valid paths are very alike the training data. Peeks in the invalid paths don’t correspond to peaks in the training data.

B. Results without junk data

When the model was run on verification data generated by VI-B. As seen from the graph in Figure 3 the models perform worse the more bytes are looked at. The models perform poorly for anything other than 2 byte of look ahead. Adding more bytes to look at seems to confuse the models.

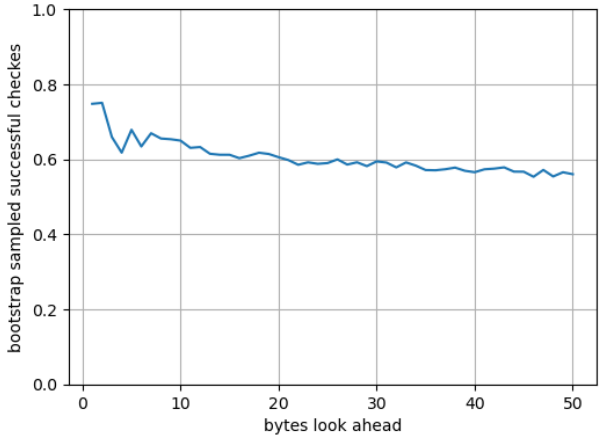

Fig. 3. Odds for path pairs to be labeled correctly. That is the invalid path was seen as ”more odd”. For very short byte look ahead – 1 to 2 bytes – the model works. For all other byte look ahead, the model performs very poorly.

It is therefore possible that more data makes the model more uncertain. The models seem to have a difficult time differentiating between relevant and irrelevant information.

Peaks in the both version of the verification data corresponding to peaks in the training data. This implies the verification data is more alike the training data.

V. Conclusion

Both of these tests have favorable conditions for isolation forest. Both models performed poorly – except when comparing 1 uniformly generated random byte. Both of the models had problems with longer byte look ahead. Some thoughts on why the models don’t perform well are listed in III.

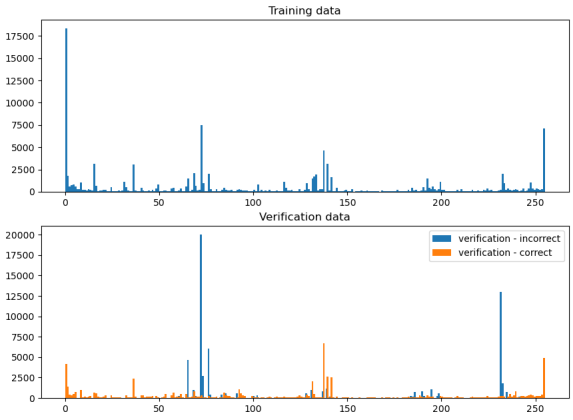

Fig. 4. Histogram over the first bytes in the tranining data and the verification data generated without junk bytes.

Isolation Forest as a model does not seem fit for this kind of machine code classification.

When the faulty branches contain randomly generated bytes the models might perform well for short byte look ahead. This performance is not due to isolation forests ability to generalize – but more likely statistical properties of the first bytes of the synthetic data. That is, the model knows how to distinguish noise from a single x86 instruction. Reasoning about sets of instructions requires more reasoning power than isolation forest seams to have.

This work gave Isolation Forest favorable conditions – and the technique struggles. If the method has a chance of working it requires a more complex model. Either more ”heavy duty” machine learning techniques or something in conjunction with isolation forest, maybe even reasoning on the level of single instructions.

References

[1] Dennis Andriesse **andothers**. *An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries*. **april** 2022. URL: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_andriesse.pdf.

[2] *curl*. **april** 2022. URL: https://github.com/curl/curl.

[3] Zhi-Hua Zhou Fei Tony Liu Kai Ming Ting. *Isolation Forest*. 2007. URL: https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/icdm08b.pdf?q=isolation-forest.

[4] *Git - fast, scalable, distributed revision control system*. **april** 2022. URL: https://github.com/git/git.

[5] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. **december** 2021.

[6] Ulf Kargén. *desync-cc — Automatic Disassembly Desynchronization Obfuscator*. **april** 2022. URL: https://github.com/UlfKargen/desync-cc.

[7] *Large single compilation-unit C programs*. **april** 2022. URL: https://people.csail.mit.edu/smcc/projects/single-file-programs/.

[8] Nikola Milosevic, Ali Dehghantanha **and** Kim-Kwang Raymond Choo. *Machine learning aided Android malware classification*. 2017. DOI: https://doi.org/10.1016/j.compeleceng.2017.02.013. URL: https://www.sciencedirect.com/science/article/pii/S0045790617303087.

[9] *scikit-learn Machine Learning in Python*. **april** 2022. URL: https://scikit-learn.org/stable/index.html.

[10] *sklearn.ensemble.IsolationForest¶*. **april** 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html.

[11] Yu-Jye Tung. *The Return Of Disassembly Desynchronization*. **may** 2022. URL: https://yellowbyte.github.io/blog/2018/the_return_of_disassembly_desynchronization/.

[12] Yu-Jye Tung **and** Ian G Harris. **?**A Heuristic Approach to Detect Opaque Predicates that Disrupt Static Disassembly**?** **in**(2020): URL: http://archive.bar/pdfs/bar2020-preprint4.pdf.

[13] *Ubuntu*. **april** 2022. URL: https://ubuntu.com/.

[14] Daniele Ucci, Leonardo Aniello **and** Roberto Baldoni. *Survey of machine learning techniques for malware analysis*. 2019. DOI: https://doi.org/10.1016/j.cose.2018.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0167404818303808.

[15] witscad.com. *Instruction Set Architecture : Instructions and Formats*. **april** 2022. URL: https://witscad.com/course/computer-architecture/chapter/isa-formats.

VI. Appendix

A. *config.cfg – with injected junk bytes*

```
verbose                     false
print_assembly              false

use_spilling                true

dry_run                     false
seed                        0
instruction_pattern         .*

junk_length_distribution    uniform
junk_length_min             4
junk_length_max             4

interval                    50
predicate_file              predicates/basic.txt
predicate_pattern           xor.*
predicate_distribution      uniform

always_taken_fraction       1.0
```

B. *config.cfg – without injected junk bytes*

```
verbose                     false
print_assembly              false

use_spilling                true

dry_run                     false
seed                        0
instruction_pattern         .*

junk_length_distribution    uniform
junk_length_min             4
junk_length_max             4

interval                    50
predicate_file              predicates/basic.txt
predicate_pattern           xor.*
predicate_distribution      uniform

always_taken_fraction       0.0
```