Machine Code Metamorphism using Recurrent Neural Networks

Antoine Champion
Email: antoine.champion@outlook.fr

***Abstract*—Software obfuscation aims to protect the intellectual property of a computer program by preventing reverse engineering. Code metamorphism is a dynamic approach of obfuscation in which a program refactors itself at run time while preserving its semantics. Surveys report that metamorphic software is more resilient to both automated and human-driven analysis in comparison with usual obfuscation methods such as encoding or encryption. However, existing metamorphic engines are heuristic and mutate the code in a deterministic manner. To address this issue, this paper presents a new metamorphic framework based on a predictive model that was trained in order to generate finite sequences of machine code instructions given a specific program context. In contrast with a typical context-agnostic metamorphic framework which would restructure specific chunks in the code, the presented model can rewrite any sequence of instructions in order to fit a specific context in the computer program. This model reached an accuracy of 99.86% on the prediction of new sequences of instructions, with an inference time of 50ms per sequence. The design and the architecture of the model are presented, methods to integrate the model within a turn-key solution are proposed, and areas of improvement are further discussed.**

***Index Terms*—metamorphism, code mutation, machine code, deep learning, recurrent neural networks.**

I. Introduction

Unprotected programs leave organizations open to piracy and technology stealing. Attackers use reverse engineering tools known as disassemblers to convert a binary executable back into assembly code, in order to find vulnerabilities, to bypass security protocols or to understand the underlying logic of the program. Several anti-tamper methods can be employed to discourage reverse engineering [1], [2]. Such obfuscators artificially increase the structural entropy[1] of a program, making it difficult to understand through human-driven or automated analysis. Unfortunatly, protecting a program from attackers without impairing its behavior is notoriously difficult [4]. While obfuscation may be immune to static disassemblers, what emerges from this process often comes with an increased computational cost and a loss of flexibility. Moreover, once the code has been deobfuscated, there are no barriers left to the attacker.

Metamorphism is a dynamic solution to obfuscation. A metamorphic program will change its internal structure at each run, while keeping the same functionalities. As it is constantly refactoring itself, it is intricate to understand using reverse engineering. Metamorphic software has also found defensive value for attack mitigation by making known vulnerabilities impossible to exploit in a consistent way [5].

There are two distinct approaches for metamorphic code generation. First, the high-level source code of a program can be dynamically generated or mutated. It either implies meta-programming and templating frameworks[2], computational reflection[3], or using a specialized language to generate the source code. A notable example is the FFTW library for computing the discrete Fourier transform, which is dynamically generating performance-critical C code using Objective Caml [8]. In the second approach, the mutations are not generated through high-level source code but directly in machine code. The program makes bitwise modifications to itself at runtime, either in memory or in its binary file. This method is often used by computer malwares in order to hide from the signature detection of antiviruses but can also prevent reverse engineering. Machine code metamorphism can be achieved through a broad range of methods: from garbage code insertion and instructions or registers swapping [9], to more advanced techniques such as structural reordering [10], [11]. Since the aforementionned methods are mutating the machine code with a certain amount of determinism, a well trained heuristic agent could detect and prevent the mutations. Ideally, a perfect metamorphic engine would receive any piece of code of variable length and would tranform it into a roughly equivalent code using a completely different set of instructions.

Therefore, the main goal of this paper is to leverage the recent advances in the field of deep learning and to use the black box predictions of a deep neural network as an non-deterministic added value, in order to build a predictive model that can rewrite any finite sequence of instructions of machine code into another functionally equivalent sequence. This paper is organized as follows. Section II provides a formal description on the mutation of machine code through a predictive model, as well as its practical limitations. Section III exposes a technical implementation of this framework, from the aggregation of the dataset to the model architecture. In Section IV, the results of the predictive model are presented and further discussed.

[1]The entropy of a program is usually measured using the Shannon Entropy or its asymptotic equivalent Kolmogorov complexity [3].

[2]Meta-programming is a programming technique which makes extensive use of the type-system of a language to execute a desired algorithm at compile time [6].

[3]Reflection is a way to retrieve and modify a program at run time. It is most often found in bytecode-compiled languages [7].

II. Formal Framework

The environment of this study depends on both a processor architecture and a program file format. While this work can be transposed to different technologies, it has been conducted on the most common architecture and executable file format association, respectively the x86 instruction set for the IA-32 architecture from Intel [12] and the Win32 Portable Executable from Microsoft [13].

Each x86 instruction in assembly language is the combination of an operation code (OP-Code) and several arguments, which get compiled into machine code. For instance, the instruction `MOV EAX, EBX` would be compiled as `89 D8`. The OP-Code for the instruction *move a 32 bits register into another 32 bits register* is `89`, and the arguments would be the two registers, respectively represented as `D` and `8`.

A. Problem Formulation

During its execution, a program has a state composed of the processor registers, the stack, the heap, and other data segments that are dynamically mapped in memory [14]. Let $\mathcal{E}$ denote the space of the states of a program, and $E_0, E_1 \in \mathcal{E}$. The goal of any instruction $i : \mathcal{E} \to \mathcal{E}$ is to alter the current state of a program into a new different state. Since the instructions are executed sequentially as a Markov process, it is possible to infer any fitting sequence of instructions that would alter any state E_0 into a new state E_1 (fig. 1).

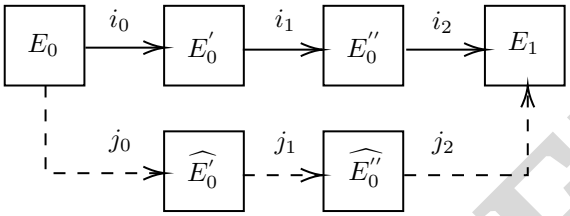

Fig. 1: Different sequences of instructions (i_n) and (j_m) can alter any state E_0 into the same destination state E_1.

Let $n \in \mathbb{N}^\star$, and $(i_k)_{k\in[[0,n]]}$ with $\forall k, i_k : \mathcal{E} \to \mathcal{E}$ a finite sequence of n instructions that alters E_0 into E_1. The goal of this metamorphic engine is to find $(j_k)_{k\in[[0,m]]}, m \in \mathbb{N}^\star$, such as :

$$\begin{cases} (j_0 \circ \ldots \circ j_m)(E_0) = (i_0 \circ \ldots \circ i_n)(E_0) = E_1 \\ \exists k, (i_k) \neq (j_k) \end{cases}$$

For each sequence of instructions (i_n) to be replaced, another sequence (j_m) that alters the same initial state into the same final state must be found. In practice, $m = n$ is needed because replacing a sequence of instructions with another sequence of different length is a complex process, due to the changes it induces in offset addressing. However, there exist an identity operation in $\mathcal{E}$, known as the `NOP` instruction (OP-Code `0x90`). Hence, in the case where $m < n$ the sequence can be padded with `NOP` instructions.

B. Reduced program state

Capturing the whole state of a program between two instructions is unrealistic: several megabytes of memory would have to be processed at each iteration of the metamorphic engine. To reduce the amount of parameters, a restricted space $\mathcal{E}_r \subset \mathcal{E}$ has been used.

The heap and the segments are memory sections that require direct addressing to be accessed: instructions are interacting with a pointer to a variable instead of its content. Including these sections in $\mathcal{E}_r$ would either require to store and process the addresses in pair with the variables in our model, or to make a direct object evaluation inducing an unrealistic overhead. Hence, such memory sections have been discarded in $\mathcal{E}_r$.[4]

The stack is a last-in-first-out data structure that can't be accessed in place. Let $S \in \{0,1\}^{N\times 32}$ describe a stack of N variables made of 32 bits each. If the last element is removed from the stack (pop), then the new stack S' can be defined as an upshift of S: $S'_{i,j} = S_{i+1,j}$. At the contrary, if a new 32-bits variable $v \in \{0,1\}^{32}$ is added (push), then $S'_{i,j} = S_{i-1,j}$ and $S'_{0,j} = v_j$. Let $U = \delta_{i+1,j} \in \mathbb{R}^{N\times N}$ denote the upper shift matrix with δ the Kronecker delta. A stack pop can then be represented as $U \cdot S$ and a stack push as $U^\top \cdot S$ without taking v into account. Using ρ as the Pearson product-moment correlation coefficient, two new parameters can be defined in the model:

$$\begin{cases} c_{\texttt{POP}} = \rho_{S',U\cdot S} = \dfrac{\operatorname{cov}(S', U\cdot S)}{\sigma_{S'}\sigma_{U\cdot S}} \\ c_{\texttt{PUSH}} = \rho_{S',U^\top\cdot S} = \dfrac{\operatorname{cov}(S', U^\top\cdot S)}{\sigma_{S'}\sigma_{U^\top\cdot S}} \end{cases}$$

Then, $c_{\texttt{POP}}$ will be close to 1 in case of a stack pop, and $c_{\texttt{PUSH}}$ will be close to 1 in case of a stack push. Since the stack of a program is usually large, keeping only $c_{\texttt{POP}}$, $c_{\texttt{PUSH}}$ and the top of the stack is enough to describe this data structure with only a few parameters in our model.

The state of a program finally includes the processor registers. Segment and pointer registers are interacting with memory addresses and won't be observed. All the registers bound to a processor extension such as SIMD are too specific and subtle to be captured. Hence, only the general purpose registers and the flag register will be observed.

Finally, $\mathcal{E}_r$ is made of the following observations: the first 3 variables on the stack, $c_{\texttt{POP}}$, $c_{\texttt{PUSH}}$ and registers `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI` and `EFLAGS`. It is equinumerous to $\{0,1\}^{322}$, a vector of parameters containing all the bits of the aforementioned observations. To fit this reduced space, the only instructions to be captured are $i_r : \mathcal{E}_r \to \mathcal{E}_r$. Therefore, instructions with direct addressing, function calls and returns, jumps and x86 extensions (SSE, AVX...) will be discarded in the collected dataset.

[4]Since different threads in a single program share the same stack but a different heap, this decision also makes the study relevant to multi-threaded programs.

III. Design and Implementation

The goal is to gather a training dataset D made of sequences of instructions of maximum length T along with their effects on the state of a program:

$$D = \left\{ (i_T), (E_{T+1}) \mid \forall k \leq T, i_k(E_k) = E_{k+1} \right\}$$

This dataset will be used to train a recurrent neural network N to output $(i_T) = \mathrm{N}((E_{T+1}))$. The complete workflow from data aggregation to model inference is illustrated in figure 2 and contains the following steps :

1) Execute a program step by step within a debugger and capture sequences of instructions along with the states between the instructions.
2) Clean, tidy and prepare the gathered data.
3) Stream the data into a Redis database and create input and output datasets for the predictive model.
4) Train a neural network using the Redis datasets.

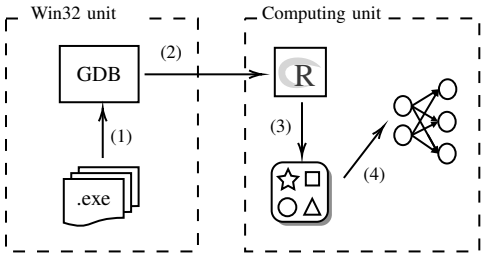

Fig. 2: The proposed workflow for a practical implementation of the framework.

The source code for all the aforementioned steps has been documented and made available on GitHub.[5]

A. Data aggregation

The GNU Project Debugger[6] (GDB) was used to execute various programs instruction by instruction while observing their state. This process was automated using the Python bindings for GDB. To avoid being stuck in event pumps or in low level loops, GDB will resume the program flow if a recurring pattern of instructions is repeated too many times. To keep the control of a program after that its flow was resumed, breakpoints needs to be placed at the entry of every function, hence the programs need to be compiled with debug symbols. This process allowed to gather a dataset of roughly 200.000 sequences of states and instructions, from a corpus of a dozen open-source programs.

B. Data preparation

For every entered function, the GDB script outputs a semi-structured JSON file which encapsulates sequences of instructions, both in NASM and in hexadecimal format, along with all of the 32-bits hexadecimal variables from the captured states. Hence, this data must be consolidated into tensors that will be served as parameters and outputs to train the predictive model. Using the `dplyr` framework for the R programming language to tidy the data [15], an input tensor $X_m^{(n)<T_x>}$ of m sequences, T_x states by sequence and n parameters has been created, along with an output tensor $Y_m^{<T_y>}$ of the same m sequences and $T_y = T_x - 1$ instructions by sequence. Each input parameter $X_k^{(i)<t>}$ is a different bit from one of the state variables. Consistency of the gathered data has then been examined through a visualization of the collected program states as shown in figure 3.

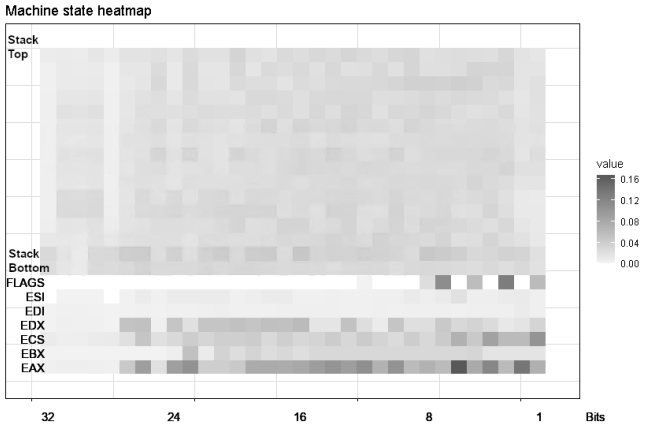

Fig. 3: Heatmap of the program state activity.

Each row of this heatmap pictures either a register or a stack variable. The n[th] column represents the n[th] bit of the observed variable using a little-endian encoding. The more times that bit has been used by the captured instructions, the darker the color. Hence, the normalized color intensities can be computed as follows:

$$H = \frac{1}{m\ T_x} \sum_{i=1}^{m} \sum_{t=1}^{T_x - 1} |X_i^{<t+1>} - X_i^{<t>}|$$

This heatmap depicts an homogeneous stack usage column-wise: every push or pop instruction will shift the whole stack up or down. The registers `EDI` and `ESI` are most of the time used to store addresses. Hence, by our scrapping rules they are barely not used. Next, the 2[nd], 4[th] and 6[th] bits of `EFLAGS` are reserved in the x86 architecture, and are not used at all. Same goes for all `EFLAGS` bits whose index is greater than 8. As a whole, this heatmap reflects the x86 architecture and the data looks consistent, so it can be further exploited.

However, the dataset is large of several gigabytes. A first iteration of model was trained on disk, but the I/O interruptions were by far the limiting factor that negatively impacted the training duration. As a result, the dataset was loaded in-memory into a Redis[7] database. Redis set, an unordered data structure optimized for random access, was used to stream data in order to train the model.

[5]GitHub repository: https://github.com/antoinechampion/DeepMetaMorph

[6]GNU Project Debugger: https://www.gnu.org/software/gdb/. To run this debugger on the Win32 platform, the MinGW compatibility layer has to be installed, along with the *gdb-python27* package.

[7]Redis: https://redis.io/

C. Predictive model

The model has to estimate the conditional probability $p(i_0, \cdots, i_{Ty} | E_0, \cdots, E_{Tx})$. If the predited probability for a whole sequence (i_{Ty}) is judged satisfying, we can safely assume that it can be used to replace the existing sequence in machine code, with the goal of transistioning from E_0 to E_{Tx}. The choice for a model architecture was based on several criterias coming from the gathered dataset:

- Inputs and outputs are sequences, hence the need for a recurrent model.
- The large amount of samples better suits a neural architecture.
- The median sequence length is 4, which allows a simple recurrent model without any attention mechanisms to handle such short sequences properly.
- Input and output sequences have different lengths with non-monotonic relationship, so a sequence to sequence (seq2seq) model should be used. [16]

Based on those criterias, an encoder-decoder LSTM using the *seq2seq* inference model has been implemented as shown in figure 4.

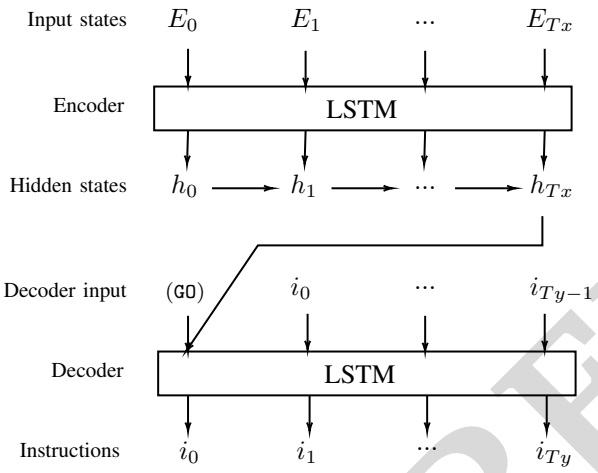

Fig. 4: Representation of the predictive model.

A first layer of LSTM cells encodes an input sequence of variable length into a fixed-dimensional representation h_{Tx}. The latter is made of both the memory state and the carry state from the last LSTM pass on the input sequence. Given h_{Tx}, a second batch of LSTM cells are predicting a fitting sequence of instructions. During the training stage, both the encoder and the decoder are given inputs from the training dataset. The mispredictions of the decoder are aggregated in a loss function which is optimized iteratively using the Adam method [17]. To use the model for inference, the decoder output at step t must be given as input for the decoder at step $t + 1$.

IV. Experimental Results

The model has been built in Python using Tensorflow and Keras. Unlike in most series prediction models where the accuracy is measured element-wise in a sequence, in our case if a single instruction in a sequence is wrong then the whole sequence is wrong. Hence, the accuracy has been measured on a full sequence basis. The training of the model has been stopped when the cross-validation accuracy reached its maximum, right before it started dropping as overfitting began to appear. At this state, the accuracy on the test dataset was 99.86%.

However, the accuracy it is not the only important metric: a metamorphic framework must operate at run time, hence it must be as few resources demanding as possible. In our case, the inference of a sequence took in average 50ms on a high-end computer. Smaller LSTM cells[8] or lighter neural architectures should be explored in further studies in order to find the best compromise between accuracy and inference time.

Another opportunity for improvement resides in the data gathering methodology. Even though many instructions and program states have been harvested in a short amount of time by examinating existing programs, this data is coming with a bias. Some instructions are present only once, hence their effect is visible through a single change of state. Other instructions are made of the same OP-Code and a small variation of their numeric argument, as an index would be incremented in a loop. Further refinement would be using a dataset of judiciously crafted instructions, examinated under different states with contextual execution using a CPU emulator.

Finally, the model has to be integrated within an existing software, with the aim of predicting new instructions according to this software's contextual state. There exist a broad range of practical methods for supervising a running program and to alter its instructions. A typical implementation (cf. algorithm 1) is to attach the program to a master process and to use OS level interrupts to interact with its state.

Algorithm 1 Interact between the model and the program using OS interrupts

```
function ReadState(pid)
    regs ← trace_registers(pid)
    stack ← read_memory(regs.ESP)
    return (regs, stack)
procedure Inspect(pid)
    attach_to(pid)
    while process_running(pid) do
        wait_random_time()
        interrupt(pid)
        adr ← trace_instruction(pid)
        E_0 ← ReadState(pid)
        for i ∈ [0; T_y] do
            step_over(pid)
        E_1 ← ReadState(pid)
        (i)_{T_y} ← model_predict(E_0, E_1)
        write_to_binary_exe(adr, (i)_{T_y})
```

[8]LSTM cells of 256 units have been used in this study.

However, such an simple implementation lacks flexibility and sums the overhead due to software interrupts on top of the one coming from the predictive model. A production-ready method would involve a significant code refactoring, for instance by making the program output regular dumps of its state to a named pipe. The pipe would be read by an external agent and processed sequentially. For every state dump, the agent uses a CPU emulator to step over T_y instructions in order to find back the final state, then sends everything to the predictive model. The overhead of the agent and the model would then be separated from the applicative code, hence the agent could even be externalized to a remote computing unit.

V. Conclusion

The presented model can help obfuscating binary executables with a significant amount of structural entropy in order to protect their integrity. A novel metamorphic framework has been proposed: a predictive model that generates series of machine code instructions to make the transition from one program state to another, using a neural network made of LSTM cells. Since the whole state of a program is too big to be processed in a reasonable amount of time, a method was found in order to reduce this contextual state into a smaller representation. A first dataset was gathered from multiple open-source programs, and was used to train the model. This led to a promising accuracy score of 99.8%. Further studies are suggested: (1) building a tailored dataset to both reduce the bias in the current dataset and to improve the model ability to generalize, (2) refining the neural architecture in order to find a good compromise between accuracy and inference speed, and (3) creating an heuristic agent in charge of generating the metamorphic code on the fly along with the execution of a program.

References

[1] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[2] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 290–299.

[3] R. Mohsen, *Quantitative Measures for Code Obfuscation Security*. Imperial College London, 2016.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, May 2012.

[5] M. Stamp, "Metamorphic software for buffer overflow mitigation," in *Proceedings of the 2005 Conference on Computer Science and its Applications*, 2005.

[6] A. Gurtovoy and D. Abrahams, "The boost c++ metaprogramming library," 01 2002.

[7] P. Maes, "Concepts and experiments in computational reflection," in *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 147–155.

[8] M. Frigo, "A fast fourier transform compiler," *SIGPLAN Not.*, vol. 34, no. 5, p. 169–180, May 1999.

[9] Y. Ling and N. F. M. Sani, "Short review on metamorphic malware detection in hidden markov models," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 7, pp. 62–69, 02 2017.

[10] P. Beaucamps, "Advanced metamorphic techniques in computer viruses," in *International Conference on Computer, Electrical, and Systems Science, and Engineering*, 11 2007.

[11] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, "Software protection through dynamic code mutation," vol. 3786, 08 2005, pp. 194–206.

[12] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 01 1997.

[13] Microsoft Corporation, *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*, 02 1999.

[14] Microsoft Developer Network (MSDN). Memory management - win32 apps. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/memory/memory-management

[15] H. Wickham, "Tidy data," *Journal of Statistical Software, Articles*, vol. 59, no. 10, pp. 1–23, 2014.

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[17] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.