Which Instructions Matter the Most: A Saliency Analysis of Binary Function Embedding Models

Moritz Dannehl[*†], Samuel Valenzuela[*†‡], and Johannes Kinder[*†]
[*]*Ludwig-Maximilians-Universität München, Munich, Germany*
[†]*Munich Center for Machine Learning, Munich, Germany*
[‡]*Center for Digital Technology and Management, Munich, Germany*
{*moritz.dannehl,samuel.valenzuela,johannes.kinder*}*@lmu.de*

***Abstract*—Current deep learning models for binary code struggle with explainability, since it is often unclear which factors are important for a given output. In this paper, we apply occlusion-based saliency analysis as an explainability method to binary code embedding models. We conduct experiments on two state-of-the-art Transformer-based models that take preprocessed assembly code as input and calculate embedding vectors for each function. We show that, during training, the models learn the importance of different instructions. From the results, we observe that call instructions and the names of external call targets are important. This observation confirms the intuition that function calls significantly impact the semantics of a function and therefore should also have a large impact on its learned embedding. This motivates the need for developing model architectures that integrate stronger analysis into preprocessing to further leverage call relationships.**

***Index Terms*—reverse engineering, deep learning, explainability**

1. Introduction

The last years have seen a rapid rise in the adoption of deep learning for tasks related to binary reverse engineering [1, 2, 3, 4, 5, 6]. Just like deep models for text and images extract high level meaning from data with a human-like or superior performance, models for binary code should eventually be able to identify high-level structure and meaning similar to how a human reverse engineer would.

Still, many open questions remain before we can hope to achieve comparable performance on binary code as we have on text and images. Currently, it is unclear how exactly binary code should be presented to the model for learning, and what kind of preprocessing should be applied to the data. Almost all published approaches rely on disassemblers to decode raw machine instructions into opcodes and operands, a deterministic and clearly defined task. But there is a broad range of options for turning opcodes, registers, addresses and constants into a limited number of tokens, and for integrating results of further static analysis.

Facts obtained from static program analysis have been successfully integrated into the training of machine learning models [5, 7, 8]. For instance, CLAP [5] implements techniques to focus the model on specific instructions, in this case intraprocedural control flow. While overall performance appears promising, it ultimately remained unclear whether this focus specifically has the expected impact. Just like with deep learning in general, the model mostly operates as a black box, with no clear indicators for its reasoning. This general issue gives rise to the interest in interpretable models or explainability methods for deep learning.

Intuitively, we expect abstract knowledge about program semantics to be reflected in a sufficiently sophisticated model for binary code. For example, it is easy to convince oneself that the behavior of a function strongly depends on which other functions it calls. In particular, the behavior of a function f that does nothing but call two functions g and h depends entirely on the behavior of those two functions. Indeed, at the language level, the semantics of f and $g; h$ would be identical, a fact that is used for the common compiler optimization of inlining. As a result, we expect `call` instructions to have a comparatively large impact on function semantics at the binary level.

In this paper, we explore to which degree the learned models align with intuitive notions of semantic importance by measuring the impact of different instructions on the resulting model output. We base our analyses on explainability techniques that have been used in various other fields including computer vision [9, 10] and natural language processing [11]. We directly take inspiration from Bastings and Filippova [12] by masking individual parts of an input to a given model and measuring the cosine similarity between the original and masked input.

Our evaluation relies on CLAP [5] and JTRANS [2], as they both (i) employ a Transformer architecture, (ii) make pretrained models available, and (iii) were trained on the BinaryCorp dataset. Both models use raw assembly code as input, which, after tokenization, is fed into a Transformer that uses token and position embeddings. Both models implement specific measures to take control flow into account: in JTRANS, each pair of instructions connected by a jump shares the corresponding source token embedding and the target position embedding. CLAP adds instruction embeddings that represent which tokens belong to the same instruction and also share weights with the source jump token embedding. Using CLAP and JTRANS as the targets of our explainability experiments, we make the following contributions:

- We raise awareness for explainability methods for Transformer-based binary function embedding models. We adapt occlusion-based saliency methods from the NLP community to the domain of assembly code (§2).
- We show that, during training, models learn to discriminate between instructions (§3.2).
- We observe that `call` instructions are among the most important instructions in a function (§3.1).

2. Methodology

We now introduce our concrete methodology for analyzing the importance of instructions. We measure occlusion-based saliency [12] to determine the impact of an instruction within a given function. In contrast to earlier work by Xu et al. [13], we *mask* the instruction rather than removing it. This avoids generating atypical model inputs that may fall out of distribution. For instance, removing a call instruction from the sequence of instructions entirely can result in a function that pushes function parameters onto the stack, followed by immediately cleaning the stack again, for no reason. In contrast, CLAP and JTRANS are pre-trained with masked language modeling (MLM), among other tasks, so both models are already familiar with parts of a function being masked. Each function is represented as a sequence of instructions. For each instruction, the tokenizer splits the instruction string output by the disassembler into tokens. In order to mask an instruction, we mask all of its tokens.

CLAP and JTRANS are well suited for our experiments, as they have models and preprocessing scripts available, and are among the currently best-performing Transformer-based models. For sake of comparison, we also use untrained models, i.e., with freshly initialized weights. We use the BinaryCorp-3M dataset as it is also used by the authors of CLAP and JTRANS and perform our investigations on the test split. We deduplicate the dataset by keeping only functions with a unique opcode hash. The opcode hash avoids false negatives from differing address layouts or register allocations across function binaries.

For every unique function f in the dataset, we measure the saliency for every instruction i by calculating the cosine distance of the embedding vector of the original, unchanged function $\mathrm{M}(f)$ and the function with the given instruction masked $\mathrm{M}(f_{\mathrm{mask}(i)})$:

$$s(i) = \cos\left(\mathrm{M}(f), \mathrm{M}(f_{\mathrm{mask}(i)})\right)$$

Lastly, we rank the instructions in a function by their saliency and calculate the saliency quantile. The reason for this is that the raw saliency values are co-dependent on the length of a function: the more instructions a function consists of, the higher the cosine similarity when masking a single instruction, i.e., the lower the impact of a single instruction. Table 1 displays an example function and the

TABLE 1. EXAMPLE FUNCTION

#	Instruction	Tokens	Saliency	Rank	Sal. Quant.
1	endbr64	1	0.998	6	0.83
2	sub rsp, 8	3	0.998	5	0.67
3	mov rdi, cs:qword_243E0	8	0.965	3	0.33
4	call cs:free_ptr	6	0.498	1	0.00
5	mov cs:qword_243E0, 0	8	0.932	2	0.17
6	add rsp, 8	3	0.998	7	1.00
7	retn	1	0.982	4	0.50

saliency of its seven instructions. The most salient instruction always has rank 1 and hence the lowest saliency quantile of 0; the second most salient instruction in this case has a saliency quantile of 0.17 ($\frac{1}{6}$), and so on, until the least salient instruction at rank 7 and quantile 1.

Given the saliency quantile of each instruction, we also consider the number of tokens each instruction consists of, as a different proportion of the function will be masked depending on the instruction length. Therefore, we plot the saliency quantile dependent on the number of tokens, consequently being able to compare instruction saliencies while keeping the number of tokens fixed. Note that for CLAP models, the maximum number of tokens is 20. Thus, all instructions consisting of more than 20 tokens during tokenization will be truncated. Instructions that are encoded with that many tokens usually have their memory operands replaced by IDA Pro with the resolved string or function name that the operand points to. JTRANS simply replaces any names and addresses with “xxx” and uses no more than four tokens.

We measure the difference in average saliency quantile for each instruction i to the overall average, and compute $t = (x - \mu_i)/s$, where x represents the overall average saliency quantile, μ_i is the average saliency quantile of instruction i, and s is the standard deviation of the overall saliency quantile. For all values, we also compute whether the difference is statistically significant using a one-sample Student’s t-test.

3. Results

We collect our results in Table 2 for CLAP, Table 3 for an untrained CLAP model, and Table 4 for JTRANS, both trained and untrained. Values left blank are not significantly different, i.e. $p > 0.01$, or have a sample size less than 100. We group instructions for simplification purposes, including mapping all opcodes matching the regex `j*` to `jmp`, combining conditional and unconditional jumps.

We report the t values normalized to standard deviation of all instructions, with negative values highlighted in red, indicating that the given instruction is more important than the average of all instructions with the same number of tokens. For the sake of clarity, we restrict the table to rather common instructions. Our implementation and complete results are available online.[1]

1. https://github.com/lmu-plai/bfesaliency

TABLE 2. SALIENCY DIFFERENCES FOR CLAP

Instr.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
add			0.2	−0.5		0.2	0.1	0.1	−0.4	−0.4	−0.4	−0.3	0.2							
and			−0.5	−0.7	−0.8		−0.5		−0.7		−0.9									
call		−1.4	−0.9	−0.8	−0.1	−0.2	−0.9	−0.5	−0.8	−0.3	−1.3		−0.8	0.0	−0.3	0.2		0.4	0.4	0.1
cmp			0.0	−0.8	−0.2	−0.2	0.1	0.4	−0.2	−0.2	−0.2	−0.2	0.3	0.2	0.6	0.6	0.8			
dec		−0.8																		
div		−1.7						−0.7												
endbr64	0.4																			−0.8
imul		−0.7	−0.9	−1.1	−0.8	−0.2	−0.7		−0.6		−1.0									
inc		−0.8				0.8		−0.2		0.3										
jmp		−0.9	−0.1	−0.5	−0.4	−0.4	−1.1	−0.7	−0.4	−0.4	−1.1	−0.7	−0.7	−0.3	−0.4	−0.3				−0.3
lea				−0.9	−0.1	0.1	0.0	−0.6	−0.2	0.1	−0.4	−0.3	−0.6	−0.5	−0.7	−0.6	−0.2	−0.8	−0.5	−0.4
leave	0.7																			
lock							−0.5	0.1		0.3										
mov			0.0	0.1	0.2	0.3	0.0	0.3	0.1	0.3	0.2	0.2	0.5	0.6	0.8	0.2	0.2	−0.3	−0.1	0.3
mul		−1.1																		
neg		−1.3																		
nop	0.4					1.4		1.0		1.2										
not		−1.0																		
or			−0.1				0.3	0.8	−0.5		0.3	1.0	0.9		0.5					
pop		0.7																		
push		0.4	0.4	0.7	0.4	0.9	1.1	1.0		1.4		1.2		1.3						−0.7
retn	−0.7																			
sub			0.4	−0.6	−0.7	0.4	0.2		−0.3		−0.3									
test			0.1				−0.3		−0.6	−0.5	−0.7		0.2							
ud2	−0.7																			
xor			0.5	0.2	0.3		0.6		−0.7		−0.4									

TABLE 3. SALIENCY DIFFERENCES FOR CLAP UNTRAINED

Instr.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
add			0.0		−0.2	0.0	−0.3	−0.3	−0.1		−0.2	−0.6								
and			0.0	−0.2	−0.7		−0.4		−0.2											
call		0.2		−0.8	−0.1	0.0	−0.1	−0.1	−0.1	0.0	−0.5	−0.1	−0.1			0.0	−0.2	0.2	0.2	0.1
cmp			0.0	−0.8	−0.5	−0.6	−0.4	−0.3	−0.2	−0.3	−0.3	−0.5	−0.1	−0.3				0.7		
dec		−0.2																		
div																				
endbr64	0.1																			
imul		0.3	0.1	−0.2		−0.3	−0.6													
inc		0.2				−0.2		−0.5												
jmp		0.2	0.1	−0.5	−0.6	−0.6	−0.8	−0.6	−0.6	−0.1	−0.4								−0.2	−0.1
lea					0.7	0.1	−0.1	−0.1	−0.1	−0.2		0.1	−0.1		−0.1	−0.1	−0.2	−0.5	−0.3	−0.3
leave	−0.6																			
lock						−0.1	−0.3	−0.1												
mov			0.3	0.1	0.1	−0.1	0.1	0.1	0.0	0.1	0.0	0.3	0.1	0.1	0.1	0.5	0.5	0.1	0.4	0.1
mul																				
neg		0.1																		
nop	0.0					0.0		−0.3		−0.1										
not																				
or			−0.1		−0.3		−0.4		−0.4		−0.6		−0.6							
pop		−0.4																		
push		0.1	−0.8	−1.4	−0.5		−0.1	0.8		1.0										
retn	0.0																			
sub			−0.1	−0.4		−0.4	−0.5	−0.3	−0.2		−0.4									
test			−0.8						−0.6		−0.3		−0.5							
ud2																				
xor			−0.8	−0.4	−0.3															

TABLE 4. SALIENCY DIFFERENCES FOR JTRANS MODELS

	JTRANS				JTRANS untrained			
Instr.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
add			0.3				−0.4	
and			0.2				−0.3	
call		−0.4				0.3		
cmp			0.2				−0.6	
dec		−0.2				−0.7		
div		−0.8				−0.5		
endbr64	−0.5				0.2			
imul		−0.3	0.0	−0.3		−0.5	−0.3	−0.2
inc		0.1				−0.1		
jmp		−0.1				−0.2		
lea			−0.2				−0.6	
leave	0.5				−0.2			
lock		−0.2	0.4			−1.0	−0.9	
mov			0.0				0.4	
mul		−0.6				−0.4		
neg		−0.2				0.0		
nop	0.2	0.5			−0.6	−0.1		
not		−0.2				−0.2		
or			0.0				−0.4	
pop		1.1				−0.1		
push		0.4				−0.1		
retn	0.4				−0.1			
sub			0.2				−0.4	
test			0.0				−1.2	
ud2	−0.5				0.1			
xor			−0.1				−1.3	

3.1. The Importance of `call` Instructions

Observation: `call` *instructions have a significant impact on function embeddings.*

In both the CLAP (see Table 2) and JTRANS model (see Table 4), `call` instructions are significantly more important for the function embedding than the average instruction with the same number of tokens masked. Note that in our dataset, 80% of the functions had at least one `call` instruction.

Additionally, we observe that `call` instructions with exceptionally many tokens do not seem to be important for CLAP. These instructions correspond to cases where, during preprocessing, IDA Pro resolved very long library function names automatically. Such long names are usually due to mangled C++ library function names which are not tokenized in a meaningful way.

3.2. The Effect of Training

For both CLAP and JTRANS, we compare the averages of saliency quantiles over the number of tokens between the pretrained model and an untrained model.

Consider Figures 1 and 2, where the straight line represents the median, and the shaded area spans the first and third quartile, respectively. We can see that the untrained model has a strong dependency on the number of tokens that are masked, while the variance over the masked instruction is lower than in the pretrained model. This shows that the model has learned to distinguish between different instructions.

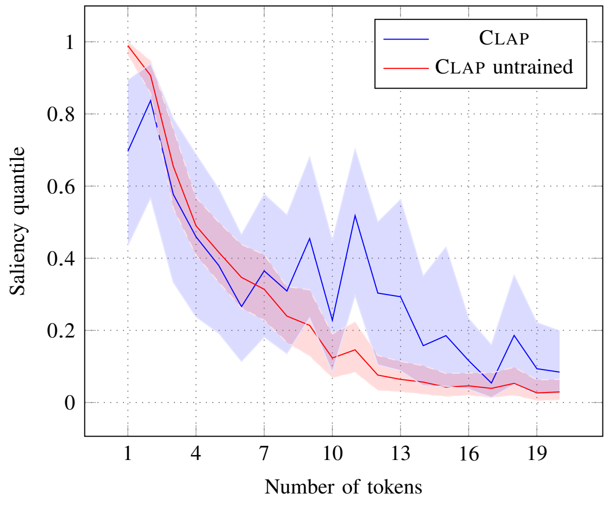

Figure 1. Saliency quantiles over number of tokens for CLAP and CLAP untrained models. Straight line denotes median, shaded area is the first and third quartile, respectively. A lower saliency quantile corresponds to higher importance.

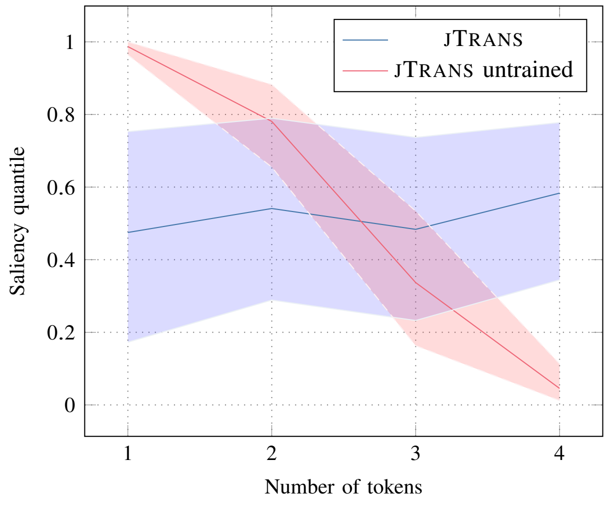

Figure 2. Saliency quantile over number of tokens for JTRANS models. Straight line denotes median, shaded area is the first and third quartile, respectively.

We also identify a correlation between the model's ability to judge the semantic impact of an instruction and its performance. We have reevaluated both CLAP and JTRANS, each pretrained and untrained, on the binary code similarity detection task using the BinaryCorp-3M test dataset, see Table 5. As expected, the pretrained variants clearly outperform the untrained models. Note that the performance of the untrained models is still far superior to random guessing, indicating that just the model architecture, including preprocessing, contributes significantly to the overall performance.

Observation: CLAP *has learned that* `endbr64` *and* `nop` *have no impact on the function semantics.*

In the untrained CLAP model, `endbr64` and `nop` have *t*-

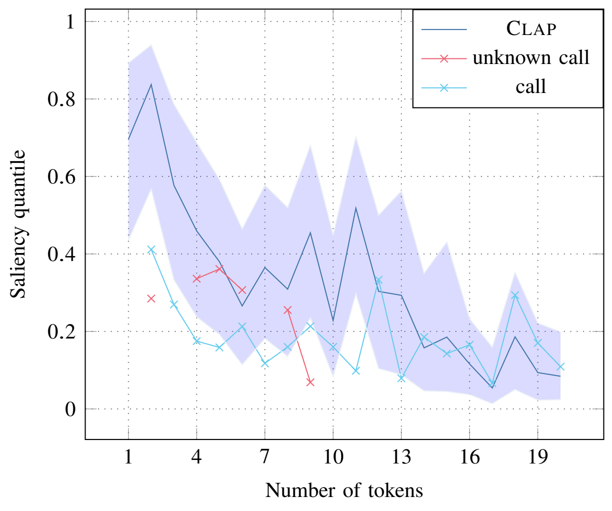

Figure 3. Saliency quantile over number of tokens for CLAP model, additionally displaying `call` instructions with unknown targets, and third party library call targets.

values of 0.1 and 0.0, respectively. In the pretrained CLAP model, both values increase to 0.4, thus indicating the lower impact on the function embedding after training.

In the pretrained CLAP model, `jmp` and `call` instructions with two tokens have a t value of -1.4 and -0.9, respectively, while the untrained model yields t values of 0.2 for both instructions, showing that CLAP has learned to pay special attention to these instructions. We observe the same effect for `call` in JTRANS (-0.4 versus 0.3), but conversely not for `jmp` (-0.1 versus -0.2).

3.3. `Call` Targets

Observation: `call` *instructions with known targets have a particularly strong impact on function embeddings.*

During preprocessing for CLAP, IDA Pro replaces addresses in calls to third-party library functions with their respective function names. In order to assess the importance of the callee function name for the call instruction, we split the `call` instructions in Figure 3 into two groups: The first, *unknown* targets, consists of all `call` instructions for which IDA Pro could not resolve the address to a function name. The second, *known* targets, contains all instructions for which a name could be resolved.

The vast majority of `call` instructions with an *unknown target* consist of four to six tokens and follow the pattern `call (reg)? (qword|sub)_[0-9A-F]*` where `reg` can be any register, and thus are direct or indirect calls to a local function. We observe a saliency quantile that is close to the average of all instructions, while `call` instructions with a known target, i.e., resolved function name, are significantly more important. Call instructions with known targets but only two tokens surprisingly appear to have low importance, but are an outlier in the dataset at only 162 samples. Most resolved calls are decorated by IDA Pro using leading underscores or segment registers, depending on the exact linking

TABLE 5. PERFORMANCE METRICS

	CLAP	CLAP untr.	JTRANS	JTRANS untr.
Recall@1	0.76	0.18	0.55	0.24
Recall@5	0.85	0.26	0.77	0.32

mechanism, which results in more than two tokens. For instance, in the dataset, `memcpy` appears only twice as `call memcpy`, but 651 times as `call _memcpy` and 4625 times as `cs:memcpy_ptr`. These differences are not resolved in preprocessing and are passed on to the tokenizer.

Longer `call` instructions with eight or more tokens that have an unknown target are indirect calls, such as `call qword ptr [r12+1358h]`. Interestingly, instructions with nine tokens have a very low saliency quantile. They differ from instructions with fewer tokens in that they have a high offset of more than two hex-digits in their memory operand, which is relatively rare. We suspect that CLAP may have put emphasis on those instructions due to their scarcity.

It is important to note that CLAP and JTRANS follow different preprocessing strategies. While most of the information obtained via IDA is retained in the input to CLAP, information such as library function names are replaced with placeholder text in the case of JTRANS. This means that not only instructions such as `mov rax, 1` are preprocessed as `mov rax, CONST`, but also instructions that CLAP recognizes as `call cs:strncmp_ptr` are converted to `call cs:xxx` during JTRANS's preprocessing. Our results suggest that this difference in preprocessing contributes to the performance gap between CLAP and JTRANS.

3.4. The Case of `endbr64`

Xu et al. [13] already investigated the impact of the `endbr64` instruction on the JTRANS model. Like its 32-bit counterpart `endbr32`, the instruction `endbr64` is a no-op used to mark valid targets for indirect control flow in control flow integrity schemes. It is only emitted under certain compiler configurations and thus can create a bias within the dataset. Removing `endbr64` led to significant improvements in JTRANS's accuracy. We can confirm the authors' findings in our experiments: for JTRANS, `endbr64` is a very important instruction, which is not the case for the untrained model. This indeed indicates a bias in the dataset.

For CLAP, `endbr64` with only one token appears to be unimportant. However, we found 629 instances of `endbr64` with 20 tokens. Although `endbr64` does not take any arguments, IDA Pro added a comment of the form `Alternative name: [...]` in those cases, which was included in preprocessing. This means that the instruction's tokens contained useful information for the model, which in turn explains why the saliency of `endbr64` with 20 tokens is very high.

4. Discussion

In our experiments, we gradually explored the relations between underlying, co-dependent variables on the saliency outcome. We started with the fact that the saliency of an instruction directly correlates with the number of instructions in a given function and thus continued with calculating the saliency quantile. Further, the number of tokens that belong to an instruction influences the resulting cosine distance which leads us to examining the saliency quantiles separately for the number of tokens masked. We suspect more effects like this that we have not yet covered. For example, so far we do not give special treatment to instructions occurring multiple times in a function.

When code is compiled with higher optimization settings, compilers begin to inline functions [14], which has a direct impact on `call` instructions. Ultimately, all of the models are trained with the objective of clustering machine code that was compiled from the same source code. Thus, the optimization setting is the distinguishing factor between instances of the same label, i.e., source code origin. Therefore, it may be interesting to investigate the impact of optimization settings on saliencies.

In the case of JTRANS, we expected `jmp` instructions to be more important than the results showed, especially because a key pre-training method of JTRANS aims to predict their corresponding jump targets. This raises the question whether its jump-awareness even contributes significantly to the performance of JTRANS. Furthermore, the `endbr64` bias severely affects JTRANS's performance, but CLAP is not affected by it. Still, we have no explanation *why* these effects occur, but we can only observe them.

Note that our analysis of the measurements is not yet exhaustive. We have focused on control flow altering instructions as well as no-op instructions such as `endbr64` and `nop`. For instance, `lea` seems to have a strong impact on the final embedding for both CLAP and JTRANS.

Despite extensive research around this topic, results of explainability techniques still often contradict each other [15] and comparing the accuracy of explainability methods is not trivial [16]. It is also not obvious which explainability technique to choose since there seems to be no universal method that performs optimally [17]. In the context of analyzing multidimensional embeddings, we decided to utilize occlusion-based saliency due to its ease of use both in terms of implementation and computing resources. Moving forward, we see value in employing additional explainability techniques or ensemble methods to gain complementary insights.

Within the field of occlusion-based saliency methods, the out-of-distribution issue mentioned previously remains relevant as well [18]. We acknowledge that systematic approaches exist to alleviate this issue altogether by modifying the explanation techniques accordingly [19]. However, since both CLAP and JTRANS are trained using Masked Language Modeling, this issue loses significance with our saliency approach.

5. Related Work

Saliency methods are post-hoc explainability tools that are commonly applied to existing machine learning models to identify which parts of an input are deemed relevant. While we only apply occlusion-based saliency methods in this paper, other prominent categories include gradient-based and propagation-based methods as well as surrogate models [12]. However, these are more challenging to apply to multidimensional representations [10] as opposed to one-dimensional predictions, which is why we do not leverage existing techniques for deep learning based security applications like LEMNA [20]. In addition to this issue, particularly established surrogate models including LIME [21] and SHAP [22] are much more computationally expensive.

While our methodology is mostly based on the occlusion-based saliency [12], our saliency computation contains additional steps, such as producing a ranked score, in order to mitigate issues that are specific to the domain of binary code analysis. In contrast to most related work, we systematically compare sections of differently sized inputs and deal with co-dependencies on underlying variables.

Occlusion-based saliency methods have also been used by Xu et al. [13] in the area of binary analysis. While the authors also measure the cosine distance between the original embedding and its perturbed counterpart, the regarded instructions are always removed completely from the function rather than being masked. Furthermore, their work focuses on detecting and deemphasizing instructions whose high saliency values stem from different compiler conventions, whereas we analyze and compare instruction significance across models and identify potential explanations for model performance beyond the model architecture or training pipeline.

6. Conclusion

In this work, we apply saliency analysis to binary embedding models. By masking out tokens for each instruction, we obtain cosine similarities in the resulting embeddings. Our statistical analysis shows that `call` instructions are among the most salient instructions, i.e., the instructions influencing the models' outcome the most for both CLAP and JTRANS, which emphasize control flow in their architectures. We encourage model developers to perform similar saliency analyses on their models to verify their models behave as expected.

Beyond that, we find that the preprocessing step of embedding binary functions likely plays a pivotal role in the model performance. Therefore, we see an important avenue of future work in regarding the preprocessing step as a separate component to the model architecture. A more detailed comparison of preprocessing strategies could provide a fruitful basis for further model development.

References

[1] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "StateFormer: fine-grained type recovery from binaries using generative state modeling," in *ACM*

SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE), 2021.

[2] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jTrans: jump-aware transformer for binary code similarity detection," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2022.

[3] J. Patrick-Evans, M. Dannehl, and J. Kinder, "XFL: Naming functions in binaries with extreme multi-label learning," in *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2023, pp. 1677–1692.

[4] H. Kim, J. Bak, K. Cho, and H. Koo, "A transformer-based function symbol name inference model from an assembly language for binary reversing," in *Proc. ACM Asia Conf. Comput. and Commun. Security (ASIA CCS)*. ACM, 2023.

[5] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "CLAP: learning transferable binary code representations with natural language supervision," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2024, pp. 503–515.

[6] T. Benoit, Y. Wang, M. Dannehl, and J. Kinder, "BLens: Contrastive captioning of binary functions using ensemble embedding," in *34th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2025.

[7] X. Li, Y. Qu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *Proc. ACM SIGSAC Conf. Computer and Commun. Security (CCS)*. ACM, 2021, pp. 3236–3251.

[8] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proc. 35th Annu. Computer Security Applications Conference (ACSAC)*. ACM, 2020, pp. 373–385.

[9] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proc. 13th European Conference Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

[10] K. K. Wickstrøm, D. J. Trosten, S. Løkse, A. Boubekki, K. Ø. Mikalsen, M. C. Kampffmeyer, and R. Jenssen, "RELAX: Representation learning explainability," *Int. J. Computer Vision*, vol. 131, no. 6, pp. 1584–1610, 2023.

[11] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv:1612.08220*, 2016.

[12] J. Bastings and K. Filippova, "The elephant in the interpretability room: Why use attention as explanation when we have saliency methods?" in *Proc. 3rd BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, 2020, pp. 149–155.

[13] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, and X. Zhang, "Improving binary code similarity transformer models by semantics-driven instruction deemphasis," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2023, pp. 1106–1118.

[14] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3561385

[15] S. Krishna, T. Han, A. Gu, S. Wu, S. Jabbari, and H. Lakkaraju, "The disagreement problem in explainable machine learning: A practitioner's perspective," *Trans. Machine Learning Research*, 2024.

[16] S. Konate, L. Lebrat, R. Santa Cruz, E. Smith, A. Bradley, C. Fookes, and O. Salvado, "A comparison of saliency methods for deep learning explainability," in *2021 Digital Image Computing: Techniques and Applications (DICTA)*. IEEE, 2021, pp. 01–08.

[17] T. Han, S. Srinivas, and H. Lakkaraju, "Which explanation should I choose? A function approximation perspective to characterizing post hoc explanations," in *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022.

[18] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du, "Explainability for large language models: A survey," *ACM Trans. Intelligent Systems and Technology*, vol. 15, no. 2, pp. 1–38, 2024.

[19] L. Qiu, Y. Yang, C. C. Cao, Y. Zheng, H. Ngai, J. Hsiao, and L. Chen, "Generating perturbation-based explanations with robustness to out-of-distribution data," in *Proc. ACM Web Conference 2022 (WWW)*, 2022, pp. 3594–3605.

[20] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. and Commun. Security (CCS)*. ACM, 2018, pp. 364–379.

[21] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?' Explaining the predictions of any classifier," in *Proc. 22nd ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1135–1144.

[22] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30 (NIPS)*, 2017, pp. 4768–4777.