Beyond Raw Bytes: Towards Large Malware Language Models

Luke Kurlandski*, Harel Berger†[1], Yin Pan*, and Matthew Wright*
* Rochester Institute of Technology, Rochester New York USA
Email: [lk3591|yin.pan|matthew.wright]@rit.edu
† Ariel University, Israel
Email: harelb@ariel.ac.il

***Abstract*—Malware poses an increasing threat to critical computing infrastructure, driving demand for more advanced detection and analysis methods. Although raw-binary malware classifiers show promise, they are limited in their capabilities and struggle with the challenges of modeling long sequences. Meanwhile, the rise of large language models (LLMs) in natural language processing showcases the power of massive, self-supervised models trained on heterogeneous datasets, offering flexible representations for numerous downstream tasks. The success behind these models is rooted in the size and quality of their training data, the expressiveness and scalability of their neural architecture, and their ability to learn from unlabeled data in a self-supervised manner.**

In this work, we take the first steps toward developing large malware language models (LMLMs), the malware analog to LLMs. We tackle the core aspects of this objective, namely, questions about data, models, pretraining, and finetuning. By pretraining a malware classification model with language modeling objectives, we were able to improve downstream performance on diverse practical malware classification tasks on average by 1.1% and up to 28.6%, indicating that these models could serve to succeed raw-binary malware classifiers.

I. Introduction

Raw-binary malware classifiers, such as MalConv [1], have gained significant attention in recent years [2], [3], [4], [5], [6]. Instead of relying on hand-engineered features extracted from the binary, these models learn representations directly from the file's raw bytes using a deep neural network (DNN). This approach is attractive for its minimal reliance on domain-specific heuristics and its ability to potentially detect subtle cues of maliciousness that handcrafted features might not be capable of representing. Yet, despite encouraging results, the raw-binary modeling paradigm has been confined to relatively small, lightweight architectures. These constraints stem from the inherent difficulty of processing extremely long sequences of bytes with neural networks and the enormous computational resources needed to store, process, and learn from large malware corpora [2]. In effect, while raw-binary classifiers hint at a future of flexible, completely autonomous malware classification, they are still far from achieving the breadth and versatility that large DNNs have offered in other domains.

The machine learning community is now witnessing a paradigm shift driven by large language models (LLMs), large neural networks trained in a self-supervised manner on massive, heterogeneous text corpora that can then be adapted to a variety of downstream tasks. The success of these models in natural language processing (NLP) is primarily rooted in the sheer scale of the models themselves [7], but also the size and quality of their training corpora. By contrast, today's raw-binary malware classifiers contain orders of magnitude fewer parameters. LLMs not only integrate diverse information sources but also learn powerful, general-purpose representations with minimal reliance on hand-crafted features or domain-specific engineering. If these benefits can be transferred to the malware domain, where each file can span millions of bytes and subtle cues of maliciousness emerge from complex global patterns, we could substantially advance the state of malware analysis.

Adapting this paradigm to the malware domain introduces unique challenges. At its core, this is a task of representation learning on code. Unlike other code representation learning tasks, however, these *large malware language models* (LMLMs) will need to 'eat' the entire program. Prior applications of DNNs for binary-code analysis, such as performing binary code similarity detection at the function level [8], can make do with examining code in smaller structured units. In contrast, malware needs to be examined holistically, as malicious behavior often arises from complex interactions between disparate blocks of code. To this end, an LMLM requires representations to be constructed from the entire program, which can be as large as hundreds of megabytes [2], far exceeding the context window of industrial-grade LLMs such as GPT-4 [9]. Since training large neural networks on long sequences is challenging, the question becomes: *How can we create large malware language models?*

Developing an LMLM consists of four main aspects: **data**, **architecture**, **pretraining**, and **finetuning**. We aim to provide insight along each of these axes.

Regarding the **data** the model is trained upon, we assess the ability of LMLMs to learn program semantics using three

[1]Most work was done during the author's time at Georgetown University.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230103
www.ndss-symposium.org

different views of a binary: raw-byte code, disassembled code, and decompiled code. In conjunction with NLP-style tokenization, we show that these approaches can reduce sequence lengths by 159x compared to simply using bytes from the beginning of the binary [1], [2], [3], [4]. While still long, this reduction in sequence length allows us to use larger, more capable model **architectures** with linear [10]—as opposed to constant [2]—time and memory complexity. When scaling such models, several works have concluded that the shape of the model is of little import compared to its size [11]. We show this belief does not hold for networks processing long sequences, an insight that allows us to use networks with 60x more non-embedding parameters than previous works [3]. After selecting two state-of-the-art architectures, we enhance them to perform both unidirectional and bidirectional sequence processing. Next, during **pretraining**, we investigate how these architectures perform at different language modeling tasks, namely, masked and causal language modeling. Malware is complex, and malicious signals may be sparse and spatially distant, so the capability of the architecture to model long-range dependencies is critical. Finally, after pretraining, we **finetune** the LMLMs for three downstream tasks: malware detection, family classification, and behavioral tagging. We not only explore the performance of different settings but also seek to determine whether or not pretraining provides tangible benefits for practical malware analysis tasks.

In this paper, we explore these questions, bridging the gap between promising raw-binary classifiers and the scale offered by LLMs. In doing so, we make several publicly available (see Appendix I) contributions to the security landscape:

- ⟨**C$_1$**⟩ We propose and compare three code-only representations of PE malware for raw-byte-style malware classifiers: bytes from executable sections of a binary file, disassembly, and decompiled code.
- ⟨**C$_2$**⟩ We show that the length of these inputs can be reduced using NLP-style tokenization by up to 7.7x. Compared to standard raw-byte classifiers, our sequences are 159x shorter, enormously reducing computation costs.
- ⟨**C$_3$**⟩ We discover and correct a profound degree of sample redundancy in PE malware datasets causing train/test leakage. Uncorrected, the redundancy can lead to overestimating classifier performance by up to 35.6%.
- ⟨**C$_4$**⟩ We conduct an architectural analysis of two state-of-the-art neural networks for long-sequence modeling, illustrating that long-sequence processing networks should be deep, not wide. Our insights allow us to use models with 60x more learnable parameters than previous work.
- ⟨**C$_5$**⟩ We pretrain a suite of LMLMs using long sequences (4x longer than the Long Range Arena [12]) for language modeling tasks. We show that high-level representations of code are easier to model than low-level ones.
- ⟨**C$_6$**⟩ We conduct extensive and diverse malware classification experiments. We demonstrate that pretraining an LMLM on language modeling can improve classification performance up to 28.6%.

II. Data Collection, Processing, & Vectorization

Data is arguably the most critical component of any machine learning model [13], [14], and especially for LLMs [11], [15]. Much of the design and usage of LMLMs will center around our ability to reliably represent malware to the model. Unlike conventional LLMs, however, the data processing pipeline for LMLMs is entirely unexplored. In this section, we explain our data collection and preprocessing approach.

A. Collection

Since Windows PE malware is by far the most widespread type of malware [16], we constrain our study to only consider PE binaries. We started with over ten million PE binaries from the Assemblage [17], Bodmas [18], and Sorel [19] datasets. We then searched a Windows 10 machine for files ending in '.exe' or '.dll' and added them to our corpus. Next, we used the Linux `file` utility to review the collected files and select samples compiled for 32-bit x86 platforms. At all stages, we carefully tracked the source and digest of each binary.

Finally, we used `diec` (Detect-It-Easy [20]) to remove samples that have been obfuscated by using encryption or compression. We remove such samples for three reasons: (i) One of our primary goals is to train statistical language models on malware. Encrypted and compressed data resembles random noise, so consequently, a model will degenerate to random guessing when exposed to such inputs. (ii) We are interested in comparing how networks learn to model the semantics of the same code represented at different levels of abstraction. These packed samples cannot be lifted correctly without de-obfuscation, so including them does not aid our objective; and (iii) We found that disassembling or decompiling these samples with Ghidra [21] could result in severely malformed outputs without warning. Unknowingly including malformed data could impact our learning experiments and the conclusions we draw from them. In reality, any real anti-virus organization (AV) would unpack [22] such software before analysis [23], as this improves the quality of extracted features [24]. We thus argue that our study approximates the status of a malware analysis pipeline after de-obfuscation.

After this procedure, we were left with just under two million files: 31K from the Assemblage corpus, 5.7K from the Bodmas corpus, 1.9M from the Sorel corpus, and 49K from the Windows machine.

B. Processing

Malware ultimately exerts its malicious effect through instructions executed at runtime, making executable code the most critical component of a binary for analysis. Note that classifiers based on control-flow [25], [26] or function-call [27], [28] graphs can accurately detect malware using only code. Other parts of a binary can often be heavily obfuscated, trivially manipulated, or even irrelevant to the core logic of the malware. Focusing just on code thus reduces the likelihood of the model learning non-robust representations [29]. It also greatly reduces the number of features that need to be considered, reducing computational cost. For these reasons, we

EXE

```
55 89 e5 53 56 57 83 7d
0c 01 75 05 e8 23 00 00
00 e8 71 03 00 00 ff 75
10 ff 75 0c ff 75 08 a1
00 30 00 10 ff d0 5f 5e
5b c9 c2 0c 00
```

DIS

```
PUSH dword ptr [EBP + 0x8]
MOV EAX,[0x10003000]
CALL EAX
POP EDI
POP ESI
POP EBX
LEAVE
RET 0xc
```

DEC

```
if (param_2 == 1) {
   FUN_100011c4();
}
FUN_10001517();
(*(code *)PTR_FUN_10003000)(
     param_1,param_2,param_3);
return;
```

Fig. 1: A subset of the same snippet of code using our three basal representations of malware.

propose training our LMLMs solely on the executable portions of binaries, in contrast to the standard practice [1], [2] of using raw bytes from the beginning of a binary (RAW).

We consider three basal representations of malware: bytes extracted from the executable sections of malware (EXE), disassembled instructions (DIS), and decompiled code (DEC) (see Appendix A for a symbol table). The EXE representation can be attained by applying very basic binary analysis. We use LIEF [30] to locate sections marked as 'executable' or 'containing code' and concatenate the bytes from these sections into a single contiguous block. DIS requires basic reverse engineering of the binary. We use Ghidra in headless mode to disassemble raw binaries. We discard the function signatures, as these require additional inference engines, and simply consider the instruction stream. DEC requires complete decompilation of the binary. Again, we use Ghidra in headless mode, this time keeping all information returned by the decompilation engine, including function names, arguments, return types, and calling conventions. Each representation offers an increasing amount of semantically dense information, but it comes at the cost of introducing more noise, since disassembly is imperfect [31] and decompilation is often convoluted [32]. Figure 1 presents an example of the output of this process.

Automating reverse engineering in this manner is far from infallible, especially when considering malware that is intentionally designed to be difficult to analyze. These challenges are compounded in the context of LMLMs, where – unlike typical LLM training – obtaining quality data can be costly, time-consuming, and error-prone. Using a combined 256 cores and 2TB of memory, our Ghidra scripts took over two weeks on our computing cluster, with the majority of this effort taken up by decompilation. In contrast, we can parse the PE headers to extract the location of the executable sections in under a day on a single machine. If the efforts required to prepare data for LMLMs using one representation are too costly or prone to misrepresentation, then there is a strong incentive to choose a different representation. Appendix B details the computational cost of attaining each representation. In brief, on a multicore server, EXE can be extracted in about a millisecond while DIS and DEC take 8 and 14 seconds/sample, respectively, on average. Weighing the benefits of using higher-level representations is a decision implementers need to make based on their ability to automate the binary analysis process. To encourage reproducibility, we release our system for cluster-scale binary analysis. The 4.11TB of binary, assembly, and decompiled code we attained using it is available upon request.

Remark 1

Preprocessing malware binaries can be either fast and easy or slow and arduous, depending on the chosen representation. Low-level representations are easier to obtain but are likely harder for models to learn from.

C. Vectorization

After attaining data (bytes and text) representing malware, we need a way to represent it numerically to the model, that is, to *vectorize* it. Many ML applications have found success representing such data as discrete *tokens* and mapping each token to a unique dense embedding vector [33]. For binary code, numerous approaches to defining a token have been proposed, including schemes that consider each opcode, mnemonic, and/or instruction to be a token [34], [8], [35], or more simply, considering each byte (BYT), as raw-byte classifiers do [1].

Given our diverse EXE, DIS, and DEC representations, rather than devise custom tokenization schemes for each, we propose to use *subword tokenization* [36] for all three. Subword tokenization begins with a coarse pre-tokenization [37] phase that breaks text into larger units called *words*. We perform pre-tokenization as follows: (i) for EXE, we model every sixteen consecutive bytes as a word; (ii) for DIS, we model each instruction as a word; and (iii) for DEC, we model each line of code as a word. After this, the larger word units are broken down into smaller components, e.g., using the byte pair encoding [36] (BPE) algorithm or unigram language models [38] (UNI).

Subword tokenization comes with several compelling benefits. First, by modeling subword units, we can circumvent out-of-vocabulary issues without turning to handcrafted normalization procedures, all while maintaining a relatively small vocabulary size. This eliminates the risk of accidentally stripping away critical information before the model has a chance to process it. Second, tokenization is a form of compression [39]; since we reserve one token for each byte/character instead of using an `[OOV]` token, this compression is lossless. This side effect is compelling for our use case, as the overwhelming length of malware has long been a barrier to the direct application of sequence learning models [2]. To our knowledge, only Rudd et al. [40] have previously considered tokenization as the vectorization step for raw-byte classifiers, but their study did not explore vocabularies larger than 396 tokens.

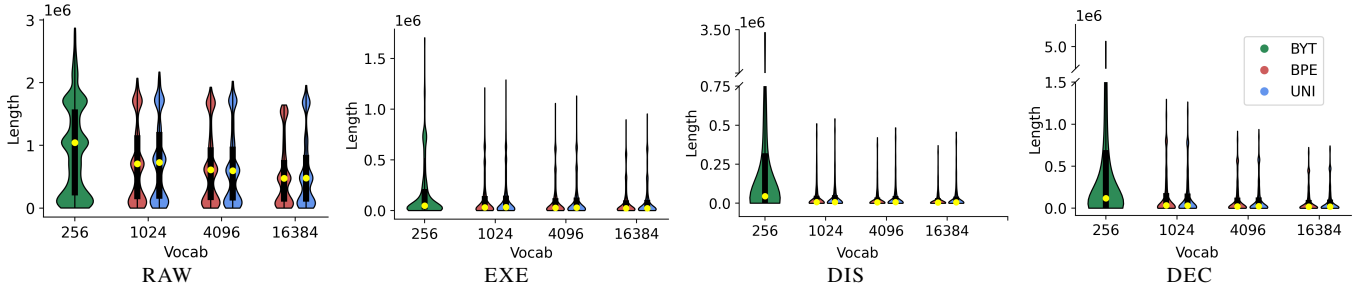


Fig. 2: Impact of vocabulary on sequence length. We trim the longest 5% of samples to make the violin plots more interpretable. The yellow dot and black bar display the median and quartiles, respectively, including the longest 5%.

The consequences of long sequences are discussed in greater detail in Section IV, but we briefly discuss some basic measurements of sequence lengths for our different input representations (EXE, DIS, DEC, and RAW) and vectorization strategies (BPE, UNI, and BYT). For each configuration, we trained a tokenizer on 5K samples (the raw-byte approach, BYT, does not require training). In Figure 2, we plot the distribution of sequence lengths after tokenization on a set of 10K samples. Appendix C provides exact compression ratios.

We see that selecting code sections from binaries greatly reduces the number of tokens that need to be processed, even before tokenizing, by up to 20.7x. After tokenizing, our code-only sequences are up to 159x shorter than those used in contemporary works [1], [2]. In general, we see that using larger vocabulary sizes results in shorter sequences, as expected, with BPE slightly edging out the UNI tokenizer. Interestingly, we see that the disassembled and decompiled code can be compressed much further than the byte-based inputs, perhaps indicating a smoothness in high-level code. Based on these results, we opt to use byte-pair encoding tokenizers with a vocabulary size of $16,384$ tokens, as these always achieve the highest compression ratios. In Appendix E, we further demonstrate that malware language models trained using this vocabulary achieve better performance than those using unigram or smaller vocabularies.

Remark 2

Code-only representations are out-of-the-box much shorter than full binaries. Tokenization is a powerful, data-driven form of lossless compression that can universally reduce malware sequence lengths even further.

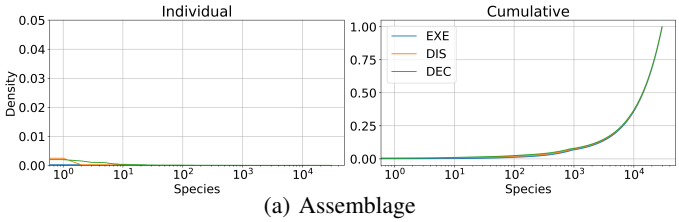
III. Reconciling Data Redundancy

Since executables commonly perform similar operations, we expect there to be a large amount of redundancy (or duplication) within samples, e.g., many C projects may contain an implementation of a linked list, so these structures might exist in many malware samples. Since our objective is to develop a representation for an entire program, we do not employ any de-duplication within each sample, as is often done when training LLMs for natural language understanding [41]. However, after preprocessing our binaries, we discovered a remarkable number of binaries have entirely identical code, as determined by comparing the MD5 digests of the binaries' EXE, DIS, and DEC representations.

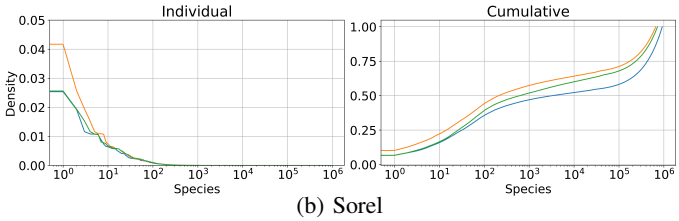
Let us define a malware *species* as a group of binaries that share identical executable code. Out of our collection of 1.9M malware samples from the Sorel corpus, we found there are only 720K unique species; moreover, 10% of the binaries from this collection belong to only four species. Contrast this with our benign corpus sampled from Assemblage, where 95% of the corpus consists of singletons. We display the full extent of this phenomenon in Figure 3. Here, we measure what we term the *individual* redundancy and the *cumulative* redundancy within the Sorel and Assemblage corpora. The individual redundancy depicts each species' proportional representation in the dataset. For example, in the Sorel plot, we see that a little more than 4% of the dataset consists of samples belonging to the most common species. The cumulative redundancy depicts the proportion of the dataset that is made up of the k most common species. In the Sorel plot, we see that just under 20% of the corpus consists of the ten most frequent species. From these figures, we see that a profound degree of redundancy exists in the malware corpus, but not in the benign corpus. Although Sorel is the largest publicly available dataset that provides PE binaries, its internal homogeneity has apparently been unknown to the community before now.

The power of LLMs arises from not only the size of the model but also the quality of the training data [42]. Therefore, we are particularly concerned about redundancy leading to train-test leakage and overly optimistic performance evaluations of malware detection systems. To mitigate this, we discard all but one sample whose decompiled code is identical. If malicious and benign samples produce identical decompiled code, both samples are discarded, as the true nature of these samples becomes ambiguous. Similarly, if multiple samples within the same malware species have differing labels (e.g., family or behavior), we retain only the sample with the most common label. These steps ensure our dataset remains heterogeneous and our evaluations are unbiased.

To assess the impact of these duplicate samples on downstream models, we conduct malware classification experiments with (Cleaned = No) and without (Cleaned = Yes) the duplicate samples present in the dataset. The uncleaned dataset represents the standard practice in the literature. The



(a) Assemblage



(b) Sorel

Fig. 3: The individual and cumulative redundancy within the Assemblage and Sorel corpora. Note that the x-axes are log scale and the species are sorted from most to least common.

Cleaned	Detection		Family		Behavior	
	ROC ↑	ACC ↑	MCC ↑	BAC ↑	JAC ↑	HAM ↓
No	0.998	0.976	0.790	0.592	0.678	0.018
Yes	0.996	0.970	0.600	0.407	0.322	0.033

TABLE I: Classification results for MalConvGCT using raw bytes (RAW and BYT) with and without the removal of redundant samples ('Cleaned'). The best value is highlighted.

details of the classification tasks are discussed at length in Section VI, but in short, we perform malware detection, family classification, and behavioral tagging, which are binary, multiclass, and multilabel tasks, respectively. We conduct these experiments using raw binaries (RAW), as this is the scenario where the leakage is almost certain to go unnoticed by practitioners. We use the state-of-the-art raw-byte malware classifier, MalConvGCT [2], with the first 1M bytes taken from the beginning of the file (BYT). The results are displayed in Table I, where we see that the model trained on the cleaned dataset achieves much lower performance than the uncleaned one, up to 35.6%. It's evident that these redundant samples can improve classification performance. The gain is illusory, however, as it overestimates the model's ability to classify truly unseen data. We hypothesize that the impact of this redundancy has inflated the performance of models proposed in previously published work, especially within the raw-binary domain, and we encourage future work to take more care to ensure that their training and testing datasets are truly disjoint.

Remark 3

Open-source PE malware datasets suffer from a proliferation of redundant, or duplicate, samples. Failure to account for this redundancy leads to train-test leakage and significantly inflates classification performance.

IV. Designing a Large Malware Language Model

With data in hand, we need to choose a neural network to learn from it. We now discuss the design of our LMLM, including its core architecture, size and shape, and directedness.

A. Backbone Neural Architecture

Training LLMs is a computationally intensive task, especially when considering long sequences. When training a neural network, there needs to be sufficient memory to handle *model states*—the optimizer, parameters, and gradients—as well as *residual states*, such as activations and temporary buffers [43]. For the model states alone, training a network with Adam [44] in mixed precision on a single GPU requires approximately 16 GB of memory for every billion parameters in the model [43]. Unlike model states, residual states are a function of the input and the computations performed within the network. While Transformers [45] are widely used as the core architecture of LLMs, the memory required for their activations scales quadratically with respect to the length of the input, making them poorly suited for our LMLMs.

To learn representations for an entire binary, we need an architecture whose space and time complexities scale linearly with the input length. Linear self-attention mechanisms [46] do this by replacing the quadratic self-attention module with a simpler, albeit less expressive, component. Another approach uses State Space Models [10] (SSMs), which are conceptually similar to Recurrent Neural Networks (RNNs), with the main difference being that the transition between hidden states is linear, making them more easily parallelized.

For our work, we've selected HRRFormer [3], which uses a clever 're-casting' of self-attention to achieve linear space and time complexity, and a recently proposed SSM called Mamba [10], which rivals Transformers at key language understanding tasks, especially on long sequences. Critically, both of these backbones offer the ability to propagate each and every input signal selectively. While effective for classifying long sequences, architectures like MalConv [2] rely on convolutions to capture and propagate information from a fixed local context, making them less capable for complex tasks where every input signal is salient, e.g., language modeling.

Like most Transformers, HRRFormer does not have any built-in concept of positionality [45]. The original authors used learned positional embeddings for the malware detection task [3], but these require an enormous number of parameters for long sequences. Since the absolute position of any token of code within a binary is mostly irrelevant to its function, it stands to reason that absolute positional embeddings provide little value. We instead implement HRRFormer with lightweight rotary positional embeddings (ROPE) [47], which do not add any additional parameters and require little additional computation. Mamba does not require a positional encoding component because of its recurrent structure, which provides implicit positional knowledge.

Using HRRFormer or Mamba blocks as a sequence processing unit, we can succinctly define our LMLM. Let $\mathbf{x} = x_1, x_2, \ldots, x_T$ be an input sequence consisting of T discrete

tokens from a vocabulary $\mathcal{V}$ with size V. First, each sequence is embedded into an H-dimensional space using a token embedding table $\mathcal{E}$. The embeddings are processed by Φ, a multilayer network consisting of L consecutive sequence processing blocks (either HRRFormer or Mamba blocks), $\phi_1, \phi_2, \ldots, \phi_L$. Due to the design of HRRFormer and Mamba, each layer ϕ has some ability to discriminately utilize information from every token in the input sequence, a property that most raw-byte malware classification approaches lack [1], [2], [4]. The latent representation of $\mathbf{x}$ is then given by $\mathbf{z} = \Phi(\mathcal{E}(\mathbf{x})) = z_1, z_2, \ldots, z_T$. Keeping other architectural details at their defaults, we are left with two hyperparameters, H and L, that determine the size and shape of the LMLM.

B. Scaling Long-Sequence Processing Networks

Even though we're using linear sequence processing algorithms, for long sequence processing, it would turn out that the memory requirements for activations far outweigh the requirements for the model states. Note that this section only outlines the key complexity terms of HRRFormer's [3] and Mamba's [10] activations; for full details, we refer the reader to the original papers as well as Korthikanti et al.'s activation analysis of Transformers [48].

In HRRFormer's attention mechanism, the parameters are dominated by the linear projections for key, query, value, and output matrices with around H^2 parameters each; the feedforward layers then contribute two linear projections with $\sim IH$ parameters each, where typically $I = 4H$ [45]. In total, we have $\sim 12H^2$ parameters per HRRFormer block. Similarly, each Mamba block's parameters are dominated by input and output projections, with about $\sim 4H^2$ and $\sim 2H^2$ parameters, respectively [10], resulting in a total of $\sim 6H^2$ parameters per Mamba block. Assuming sixteen bytes per parameter when training with Adam in mixed precision [43], the memory required for model states to train multilayer HRRFormer and Mamba networks is then approximately $\sim 192LH^2$ and $\sim 96LH^2$ bytes, respectively.

Analyzing residual states is much harder, as deep learning frameworks apply non-trivial optimizations to reduce the impact of intermediary buffers and the like. Nonetheless, concerning activations, HRRFormer requires at least: TH activations for each key, query, and value projection; TH activations for its superposition and value approximations; TH for softmax; TH for weighted values; $\sim 19TH$ for the feedforward layers [48]; and $\sim 4TH$ for layer normalization [48]. In total, a multilayer HRRFormer requires $\sim 480LTH$ bytes to process a single input with 16-bit activations. For Mamba, the largest source of activations is in the SSM expansion, where using the default SSM expansion factor $S = 16$ and intermediate size $I = 2H$ [10] results in $\sim 32TH$ activations. Accounting for the relatively small activations from other projections and RMSNorm, we can say that Mamba requires at least $\sim 35TH$ activations per block. Therefore, in a similar setup as the multilayer HRRFormer, a multilayer Mamba requires $\sim 560LTH$ bytes for activations alone.

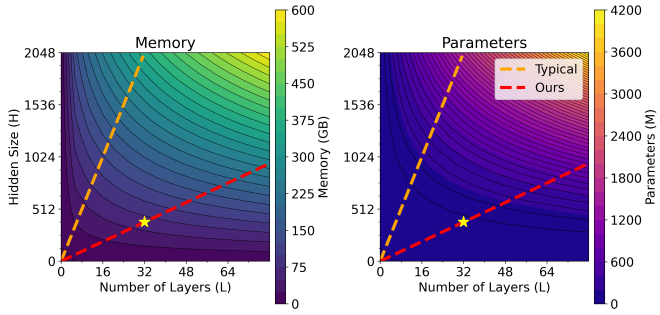
Suppose we select a shape for the LMLM similar to that of BERT [49], with $H = 768$ and $L = 12$. The memory for HRRFormer's model states is then approximately 1.27 GB. However, if learning from malware sequences that are 2^{16} tokens long (65.5K), the activation memory is much larger, at 270 GB. For reference, NVIDIA's 2024 Hopper 200 barely has half that much memory. While HRRFormer was envisioned with extreme sequence processing in mind, the authors used a very small model: $L = 1$, $H = 256$, and less than one million non-embedding parameters for malware detection. Evidently, the explosion of activation memory is a critical bottleneck when it comes to scaling malware-processing neural networks.

Across multiple domains, a body of work has demonstrated that the number of parameters in large foundational models is more important than the model's shape [7], [50], [51]. If we presume the same to be true for malware, our objective is then to maximize the parameter count in our models while minimizing computational expense, particularly the problem of handling a large number of activations. To do this, we can utilize activation recomputation [52], which roughly reduces the activation memory across layers from $\mathcal{O}(L)$ to $\mathcal{O}(\sqrt{L})$ by recomputing the activations during the backward pass instead of storing them. At the expense of around 30% additional compute time [52], using activation recomputation, the activation memory requirements for HRRFormer and Mamba are roughly given by $\sim 480\sqrt{L}TH$ and $\sim 560\sqrt{L}TH$. Returning to our example with HRRFormer, the activation memory becomes much more manageable at 77.9 GB.

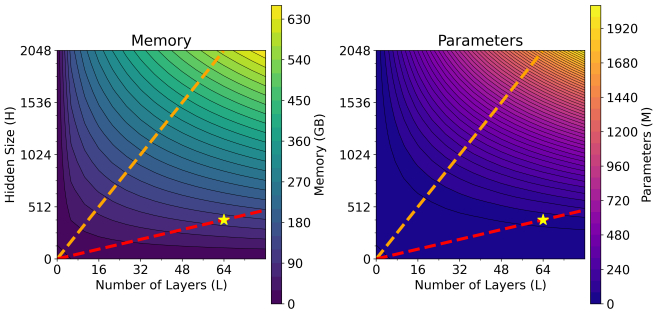
Assuming $T \gg H \gg L$, due to the sub-linear complexity of L, it's clear that the best way to increase the number of parameters in the model while modulating activation memory is using a large L and a moderate H. Modern LLMs usually have a width-to-depth ratio of around $H/L = 64$ (or $H/L = 32$ for Mamba). Given our goal of long sequence modeling, we propose to use much deeper networks with a relatively meager hidden dimension. To use similarly-sized models, we'll employ causal and bidirectional HRRFormer models with $H = 384$ and $L = 32$, causal Mamba models with $H = 384$ and $L = 64$, and bidirectional Mamba models with $H = 384$ and $L = 32$. Each of these models has roughly 60 million non-embedding parameters, making them roughly the size of smaller LLMs, e.g., ALBERT [53], which is all we are capable of training on our available hardware. Note that this is $24\times$ more learnable non-embedding parameters than the enhanced MalConvGCT architecture [2] and $60\times$ more than the original HRRFormer implementation [3], all while providing each and every layer access to each and every token (no downsampling along the temporal dimension).

We provide a visualization of this width-depth tradeoff in Figure 4 for $T = 2^{16}$. The shape of the contours in the memory plot, growing more rapidly when increasing H compared to L, compared to the shape of the contours in the parameter plot, which grow nearly equally with regard to H and L, demonstrate that it's more efficient to increase L rather than H to maximize parameter count. In each sub-figure, we draw a line representing the value of H/L commonly seen in the

literature for that model and a line representing our ratio of H/L, as well as a star indicating our chosen hyperparameters.



(a) HRRFormer



(b) Mamba

Fig. 4: Memory requirements (model states and activations) and parameter counts of HRRFormer- and Mamba-based LMLMs for long sequence processing.

Remark 4

Activation memory is the critical bottleneck when it comes to scaling malware-processing neural networks. To maximize the number of parameters in an LMLM, we recommend a deep and narrow architecture shape.

C. Architectural Modifications for Differing Directedness

Causal architectures process inputs sequentially from beginning to end, forwarding a hidden state at each time step t. Bidirectional architectures process inputs from all time steps simultaneously or process inputs from both directions. While causal language models [54] are popular in NLP for generative tasks, we are focusing on classification, so causal architectures are not required. We nevertheless believe it is worth experimenting with both causal and bidirectional LMLMs. The original implementation of HRRFormer, however, is deeply bidirectional, while Mamba is inherently unidirectional. Here, we present architectural modifications to use both of these models in a causal ($\rightarrow$) and bidirectional ($\leftrightarrow$) manner.

1) Causal Implementation of HRRFormer: HRRFormer [3] replaces the Transformer's costly self-attention module with an alternative token-mixer based on symbolic AI. The first step in HRR attention is to create a superposition (β) between key (k) and value (v) vectors using a special binding operation ($\circledast$) based on *Holographic Reduced Representations (HRRs)* [55]:

$$\beta = \sum_{i=1}^{T} k_i \circledast v_i \tag{1}$$

The binding mixes information from each key and value together; due to the properties of HRRs, the summation preserves much of that information while being extremely compact. Next, inverse ($\dagger$) query (q) vectors are bound with the superposition to create value approximations ($\hat{v}$) for every time step t:

$$\hat{v}_t = q_t^{\dagger} \circledast \beta \tag{2}$$

The cosine distance between these approximations and each original value vector is computed to create a *pseudo-attention vector*, replacing the standard dot product between keys and queries seen in vanilla attention [45]. This vector is then normalized and used to weight the value vectors.

Due to the early interaction between every key and value when forming the superposition, every step after that contains information from all time steps. There is no way of masking out illegal information from future tokens immediately before combining the attention scores with the values, as we can with self-attention [45]. Instead, to create a causal implementation of HRR attention, we create T superpositions, one for each time step, containing only interactions between keys and values associated with the tokens preceding that time step. Our superposition associated with time step t is given by:

$$\beta_t = \sum_{i=1}^{t} k_i \circledast v_i \tag{3}$$

Critically, this operation can still be computed with the same time complexity, $\mathcal{O}(TH \log H)$, as the original superposition using simple dynamic programming techniques. The space complexity required to store the superpositions increases from $\mathcal{O}(H)$ to $\mathcal{O}(TH)$, but the spatial complexity of other operations in HRR attention is already $\mathcal{O}(TH)$, so this is of little consequence. Following that, each value query is only allowed to unbind with superpositions containing information from preceding tokens (replacing β with β_t in Eq. 2):

$$\hat{v}_t = q_t^{\dagger} \circledast \beta_t \tag{4}$$

Again, this operation incurs no additional space or time complexities compared to the original. Since each value approximation is causal, the pseudo-attention scores are also causal, as is the final output of the HRR attention mechanism.

2) Bidirectional Implementation of Mamba: Our bidirectional implementation of Mamba is much simpler and heavily inspired by bidirectional RNNs. Concretely, we train two Mamba backbones concurrently, Φ_{FORW} and Φ_{BACK}, which process $\mathbf{x}_{\text{FORW}} = x_1, x_2, \ldots, x_T$ and $\mathbf{x}_{\text{BACK}} = x_T, x_{T-1} \ldots x_1$, respectively. At this point, most bidirectional RNNs would concatenate the forward and backward models' hidden states, resulting in a cumulative latent representation $\mathbf{z} = [\mathbf{z}_{\text{FORW}}, \mathbf{z}_{\text{BACK}}]$ with hidden size $2H$. To keep the hidden

size small and avoid the activation memory bottleneck, we sum them along the hidden dimension: $\mathbf{z} = [\mathbf{z}_{\text{FORW}} \oplus \mathbf{z}_{\text{BACK}}]$.

V. Learning from Unlabeled Data

As in most LLMs, we propose to pretrain our LMLMs using *self-supervised learning*. Self-supervised learning is a learning paradigm where a supervised learning objective is formulated directly from data itself without relying on externally provided labels, e.g., reconstructing perturbed input data [56]. This allows us to learn representations from large quantities of data without the need for quality labels, or any label at all. In the malware space, this is extremely valuable, since attaining accurate malware labels is difficult [57] and automatically labeling malware can be inaccurate [58].

A. Language Modeling

We pretrain our causal models on next token prediction, also referred to as causal language modeling (CLM) [54]. At each time step, the model is provided tokens from all previous time steps and tasked to predict the token at the current time step. The model parameterized by Θ is trained on the cross entropy between the model's predicted probability distribution over all tokens in the vocabulary and the ground truth value:

$$\mathcal{L}_{\text{CLM}} = \mathbb{E}_{\mathbf{x}} \sum_{i<|x|} -\log P_\Theta(x_i|x_0, x_1, \ldots, x_{i-1}) \tag{5}$$

We pretrain our bidirectional models using masked language modeling (MLM) [49]. Here, the model is presented with an input sequence and a boolean mask, $\mathbf{m}$, masking out a random $p\%$ of tokens in the input. While the original authors of MLM pretraining masked $p = 15\%$ of tokens, follow-up studies have shown that very large models can benefit from more aggressive masking, as high as 40% [59]. Based on these findings and the size of our models, we use $p = 25\%$. The model is tasked to predict the original value of the masked tokens in the same manner as the causal task. Formally, the model is trained to minimize this loss function:

$$\mathcal{L}_{\text{MLM}} = \mathbb{E}_{\mathbf{x},\mathbf{m}} \sum_{i:m_i=1} -\log P_\Theta(x_i|x_{-\mathbf{m}}) \tag{6}$$

Causal language models are commonly evaluated using perplexity (PPL), an intrinsic measurement of the model's 'surprise' over an unseen distribution. This metric has also been applied to evaluate masked language models [60]. For a sequence of tokens $\mathbf{w}$ that the model was tasked to predict, the perplexity is given by Equation 7 where $\mathbf{c}_i$ is the context given to the model for token w_i.

$$\text{PPL}(\mathbf{w}) = \exp\left[-\frac{1}{|\mathbf{w}|}\sum_{t=1}^{|\mathbf{w}|} \log P_\Theta(w_i|\mathbf{c}_i)\right] \tag{7}$$

Unfortunately, a model's perplexity is particularly impacted by the size of the vocabulary. Models with larger vocabularies need to be able to predict a more diverse set of tokens and thus will naturally be more surprised. To fairly evaluate models with different vocabularies, Tao et al. [61] propose to normalize the model's surprise on any given token by the probability of each token occurring within the given vocabulary, $P_{\mathcal{V}}(\cdot)$. The normalized perplexity (NPPL) is given by:

$$\text{NPPL}(\mathbf{w}) = \text{PPL}(\mathbf{w}) \times \exp\left[\frac{1}{|\mathbf{w}|}\sum_{t=1}^{|\mathbf{w}|} \log P_{\mathcal{V}}(w_i)\right]. \tag{8}$$

The NPPL provides future works with an opportunity for fair comparison and is critical to compare the LMLMs in our vocabulary ablation experiments (Appendix E).

B. Experiment

We train our LMLMs on the first T bytes from a little over 800k samples (filtered for redundancy). We use the AdamW optimizer from PyTorch with high weight decay ($\lambda = 0.1$) to promote generalization during finetuning. To speed up convergence, we warm the learning rate up linearly to $\gamma = 0.001$ over the first 5% of training, then decrease it linearly until training finishes. We use a microbatch size of 1 — the largest that fits on an A100 with 40GB of memory — and accumulate the gradients over 1024 samples before clipping to a norm of 1.0 and updating the model weights. We train in mixed precision with 16-bit brainfloats, but keep Mamba's residuals and HRRFormer's self-attention in 32-bit precision to prevent numerical overflow. To assess long-sequence modeling, we train models with $T = 2^{12}$ to 2^{16}, and evaluate each on a hold-out set of 4096 samples.

C. Results

The results are shown in Figure 5. We found Mamba to outperform HRRFormer across both CLM and MLM tasks on the EXE, DIS, and DEC representations, as evidenced by lower PPL and NPPL (lower is better). This is increasingly apparent for longer sequences (larger values of T). We suspect that the superposition described in Equations 1 and 3 cannot properly model interactions from all tokens when T is large relative to H, which is compounded by the deep/narrow shape.

More interestingly, we found differences between the performance of these networks modeling the EXE, DIS, and DEC representations, even though they are learning from the same binaries. In particular, the models had an easier time learning from decompiled samples than disassembled ones and an easier time learning from disassembled samples than binary ones. Despite the risk of errors in the disassembly and decompilation processes, the models learned better from higher-level representations.

Remark 5

Deep SSM-based LMLMs outperformed Transformer-based ones at language modeling, especially at longer sequence lengths, and high-level code representations of malware are easier to model than low-level ones.

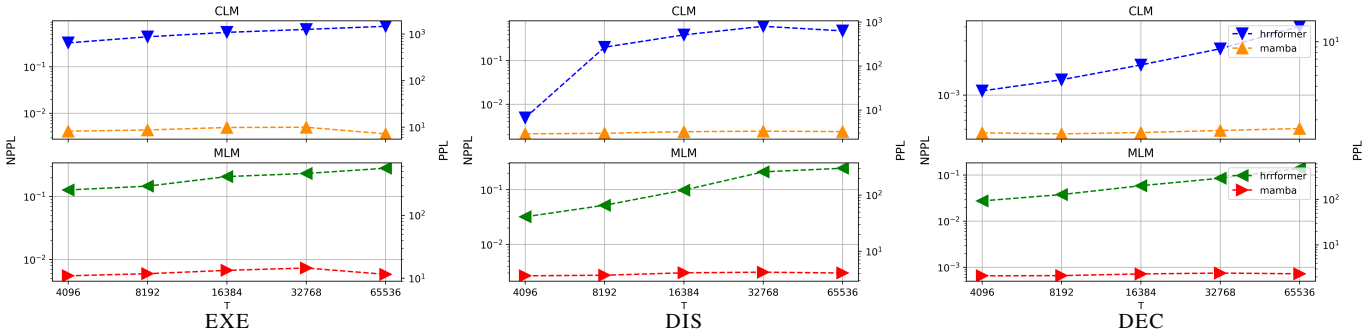


Fig. 5: Performance of causal and masked language models trained to model malware at different sequence lengths.

VI. Applications for Practical Malware Analysis

Following pretraining, we can finetune a pretrained model for downstream tasks, such as malware detection, family classification, or malware behavioral tagging.

A. Threat and System Models

We build our threat model based on how industry-grade anti-virus systems accumulate and process software binaries [23]. Malware binaries are often packed to make initial analyses less effective. We remove packed samples, but many of these can be easily unpacked (the most prevalent packer, UPX, comes with an unpacking switch), or unpacked via a growing body of sophisticated generic software unpackers [62] [63] [64]. Some malware would likely not be able to be unpacked and thus would remain inaccessible to static analysis. Finally, as our core focus is on malware modeling, we do not consider an active attacker who can modify their software in direct response to our system's behavior, creating adversarial examples [65].

For our system model, we consider a defender who performs static analysis on Windows PE executables after routine transport-layer decoding and commodity unpacking, so the model receives a complete on-disk binary but no dynamic traces, kernel hooks, or network telemetry. This sits at the stage immediately following software unpacking in multi-tier detection pipelines, where samples flow through increasingly sophisticated analyses, and lets us quantify the benefit of progressively richer static views (EXE, DIS, DEC) once a file is analyzable. Because a static classifier can inspect only the bytes present in that image, it cannot reason about threats that never materialize statically, such as downloaders that stream a DLL into memory, or samples that remain fully obfuscated by strong encryption.

Researchers have shown that the apparent success of ML on packed malware usually stems from spurious surface cues (e.g., packer stubs, anomalous import tables, high-entropy sections), rather than semantic properties [24] [66]. Requiring binaries to be unpacked into a canonical PE form, therefore, removes these questionable signals and yields evaluations that better reflect a detector's true capabilities. In Appendix G, we show that when packed and unpacked samples are re-introduced at validation time, accuracy drops on packed files but is fully restored once they are unpacked, underscoring the practical relevance of our threat and system models.

B. Leveraging Multiple Inputs

Building on our findings from pretraining, we introduce an ensemble model that leverages all three code representations simultaneously. While lifting binaries to higher-level forms simplifies structural complexity, it can also introduce noise due to the limitations of binary analysis. To balance these trade-offs, our ensemble approach processes each representation independently using an LMLM pretrained on that input type. The resulting latent vectors are then concatenated and passed through a *combiner* network (a linear layer) to produce a classification decision. Each LMLM extracts detailed latent features from the EXE, DIS, and DEC views, while the combiner learns to weigh and integrate these perspectives for downstream tasks. We refer to this multi-view input as ALL (EXE + DIS + DEC). Figure 6 illustrates the architecture.

C. Malware Detection

Malware detection is the binary task of determining whether or not a sample is malicious. This is arguably the most critical application of malware analysis, but also one of the most challenging to train machine learning systems to perform in an unbiased manner. Pendlebury et al. [67] stress that when training and evaluating machine learning malware detection systems, we should: (i) ensure that the ratio of malware in our validation sets matches the ratio of malware in the wild, and (ii) ensure that the samples in the validation sets come from a later point in time than the samples in the training set.

Several sources estimate the ratio of Android applications in the wild that are malicious to be roughly 10% [67], but to our knowledge, there are no analogous studies for PE malware. We thus conduct our experiments using balanced train and test sets with 50% malware and 50% goodware, as is done in other recent work that uses PE files [68], [5]. We provide an ablation experiment with different ratios in Appendix F. With regard to the temporal constraints, we use the timestamps provided by the original curators of the Assemblage, Bodmas, and Sorel datasets, and the Linux `objdump` tool to attain timestamps for our Windows collection. We discard samples with timestamps that are anomalous and then select only samples from 2010 to 2020. Finally, we split the files into temporally disjoint train and test sets with 25K and 8K samples, respectively, such that each split has an equal ratio of malware and goodware.

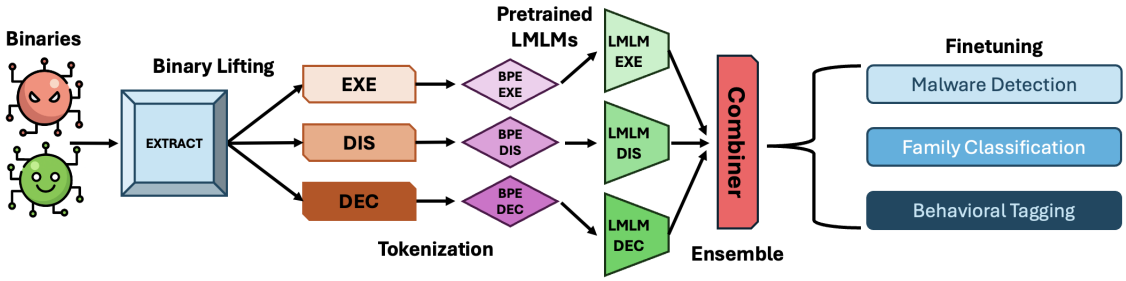


Fig. 6: Diagram of the LMLM ensemble, leveraging all three code representations simultaneously.

To evaluate models on binary malware detection, we use both the area under the receiver operating characteristic curve (ROC) and the accuracy (ACC). ROC gives a broad view of discriminative capability, while ACC provides a more intuitive measure of the classifier's capabilities.

D. Family Classification

In addition to knowing whether or not a sample is malicious or benign, knowing more about how a malware sample relates to other malware can improve an organization's incident response and understanding of its adversaries. Malware family classification is the multiclass task of determining the granular family that a malware sample belongs to. We use VirusTotal [69] to attain threat reports for our samples and then use AVClass2 [70] to consolidate the reports from an ensemble of antivirus vendors into a single classification label.

After attaining family labels, we are left with a long-tailed distribution, where some malware families are extremely common and others are extremely rare. For example, out of 262K samples (after filtering for redundancy), there were 107K samples from the 'virlock' family, while 1092 families had only a single sample. In addition to learning representations that differentiate malware families with few labeled samples, we also need to ensure that these less common classes are not ignored, given the more common ones. In this experiment, we use the simple class re-weighting approach from Cui et al. [71], as used by the MORSE authors [58], setting the weighting hyperparameter $\beta = 0.99$ (here, β does not refer to HRRFormer's superposition). Before applying the re-weighting, we remove classes with fewer than 10 samples and randomly downsample 23 especially common families such that the imbalance factor is 100, leaving us with 522 unique malware families to differentiate between. When creating the train/test split, we allocate one sample from every class to each split and then distribute the remaining samples between the sets randomly. The outcome is a dataset with 49K training samples and 13K testing samples. In the security discipline, especially, it is critical to train models using data representative of real-world distributions [72], so using a highly imbalanced dataset improves the realism of our experiments and the certainty of the conclusions we may draw from them.

To evaluate models on multiclass malware family classification, we employ the Matthews correlation coefficient (MCC) and the arithmetic mean of class recalls, also known as balanced accuracy (BAC). MCC is a single-value assessment of classification quality that is more informative than accuracy in imbalanced scenarios. BAC, on the other hand, aggregates per-class recalls, ensuring that poor performance on even a single minority class lowers the final score.

E. Behavioral Tagging

For our final task, we consider the perspective of a malware analyst: what if we encounter a new sample of malware that falls outside of known families? We may know that it is malware, but how does it work? *Behavioral tagging* is the process of determining which malicious behaviors a sample exhibits. We get behavioral tags from the same VirusTotal reports we used for family classification. We process the reports using ClarAVy [73], which fills a similar role as AVClass2, except that it aggregates behavioral tags across AV vendors rather than family labels. For each sample, we consider behaviors identified by at least two different AVs as valid labels.

For each behavior, the goal of the classifier is to assess whether or not a sample exhibits that behavior. The classification of each behavior is thus binary, but there are dozens of behaviors to test. Since most malware only exhibits a small number of behaviors, there is a strong bias towards negative labels within the dataset. This general property of multilabel classification often pushes models to be overly cautious when predicting the presence of a label [74]. To promote the identification of behaviors, i.e., encourage the model to predict that a sample exhibits some behavior, we use focal loss [75]. Compared to binary cross entropy, focal loss dynamically weights the loss per sample to provide greater learning value to samples that are classified with low confidence. We use the hyperparameters proposed by Ross et al. ($\alpha = 0.25, \gamma = 2.0$) [75].

In addition to coping with a disproportionate amount of negative labels, multilabel classification also results in an extremely complex label surface, since the number of possible labels scales exponentially with the number of classes. To reduce the complexity somewhat, we only include the 50 malware behaviors with at least 100 representative samples, and we downsample samples from particularly common classes to attain an approximate imbalance factor of 10. We create train and test splits using the same approach as for family classification, resulting in 13K training and 3K testing samples.

To evaluate models on multilabel behavioral tagging, we use the Jaccard index (JAC) and Hamming loss (HAM). JAC, also known as intersection over union [76], measures the overlap between predicted and actual sets of labels, computed as the size of their intersection divided by the size of their union. HAM quantifies the fraction of incorrectly predicted labels. These metrics provide complementary views: the Jaccard index focuses on how well the predicted set matches the true set as a whole, while Hamming loss assesses the rate of errors.

F. Experiment

For each task, we compare each finetuned model against a network trained from scratch with the same architecture as well as the MalConvGCT (MalConv) architecture from [2]. We train every classifier for five epochs with the same learning rate scheduler we used for pretraining (Section V-B). We use a peak learning rate of 10^{-3} for MalConv, 10^{-4} when training HRRFormer and Mamba from scratch, and 10^{-5} when finetuning the pretrained HRRFormer and Mamba models (models trained from scratch benefit from higher learning rates than those being finetuned). We use AdamW with the typical weight decay ($\lambda = 0.01$) and a minibatch size of 64 using gradient accumulation. Each model is trained in mixed precision as done in pretraining, except MalConv, which is lightweight enough to train in full precision. Again, all samples in these datasets have been filtered for redundancy (Section III). Appendix B details our comparison of each architecture's computational cost.

G. Results

The classification experiment results are displayed in Table II, highlighting performance differences across inputs, architectures, directedness, and pretraining. We now discuss trends among each of these aspects in detail. To aid this discussion, Table III displays the average performance of the models within each of these experimental parameters.

1) Input: The difference in classification performance between EXE, DIS, and DEC representations is less significant than it was for language modeling. For malware detection and behavioral tagging, leveraging ALL inputs tends to result in the best performance. Family classification seems to work fine with EXE or DEC.

2) Architecture: While competitive at binary malware classification, large, deep, and narrow HRRFormer configurations were not useful for complex malware analysis tasks like family classification and behavioral tagging. In many cases, the model degenerated to essentially random guessing or refusing to identify any behaviors. At any given task, the best Mamba-based models beat both HRRFormer and MalConvGCT.

3) Directedness: Although a few counterexamples exist, bidirectional models typically outperform their unidirectional counterparts by a substantial margin. For example, bidirectional Mamba pretrained on the combined ALL representation has a 0.454 Jaccard index for behavioral tagging, while the unidirectional model only achieves 0.382. Based on the average trends, bidirectional context appears to be valuable for all classification tasks and all input representations.

4) Pretrained: Pretrained models outperform their counterparts trained from scratch more often than not, on average by 1.1% across all tasks and metrics. The largest difference was for the unidirectional HRRFormer on EXE, which was 28.6% more accurate when pretrained. On average, pretraining is more beneficial for malware detection and behavioral tagging than it is for family classification, and furthermore, is more beneficial for models using DEC and ALL inputs than those using EXE and DIS inputs. Most importantly, unidirectional models tend to benefit more from pretraining than their bidirectional counterparts.

H. Computational Costs

The time required for a forward pass during inference per sample for each HRRFormer/Mamba is roughly similar — the fastest architecture (bidirectional HRRFormer) is 75.8% faster than the slowest architecture (bidirectional Mamba). By contrast, the difference between MalConv and the LMLM-based architectures is more extreme, being between 3.82x and 6.72x faster. As a rule of thumb, we allocated an order of magnitude more GPU time to pretraining than classification.

Remark 6

Pretrained LMLMs usually outperform the same model trained from scratch. Malware classifiers can benefit from differing views of the same file, i.e., using byte-level code, disassembly, and decompiled code together.

VII. Discussion

While LMLMs can learn sophisticated patterns from code-only representations of malware, obtaining high-quality data is nontrivial for some of the proposed input representations. Reverse engineering pipelines remain fragile, computationally expensive processes that depend on expert systems engineers. To some degree, this reintroduces the human into the detection workflow, an aspect of malware analysis that raw-byte classifiers had hoped to eliminate. Nonetheless, simply feeding an entire disassembly file to a model requires far less human involvement than developing manually engineered feature sets such as the Ember features [77].

Despite advancements, LMLMs continue to exhibit limitations intrinsic to static analysis, such as vulnerability to obfuscation techniques like reflection and packing [24]. Rather than replacing other techniques, LMLMs can be a useful addition to a comprehensive malware classification system that incorporates heuristic- as well as learning-based identification methodologies using both static and dynamic analysis. More broadly, LMLMs and other ML-based malware detection methods degrade over time and need frequent retraining with the latest strains of malware to remain performant [78][63].

For future work, several major directions stand out. In terms of data, a deeper exploration of vocabulary formation could substantially impact the compression efficiency, expressiveness, and flexibility of LMLMs. For the model, alternative pretraining methodologies beyond causal or masked language

Input	Architecture	Directedness	Pretrained	Detection		Family		Behavior	
				ROC ↑	ACC ↑	MCC ↑	BAC ↑	JAC ↑	HAM ↓
EXE	HRRFr	→	✓	0.901	0.571	0.049	0.007	0.000	0.056
		→	✗	0.988	0.808	0.077	0.011	0.000	0.056
		↔	✓	0.977	0.699	0.208	0.028	0.003	0.055
		↔	✗	0.977	0.898	0.284	0.041	0.000	0.056
	Mamba	→	✓	0.995	**0.973**	0.558	0.346	0.381	0.033
		→	✗	0.962	0.902	0.412	0.286	0.296	0.040
		↔	✓	**0.996**	0.968	0.613	0.399	0.356	0.033
		↔	✗	0.966	0.898	0.585	**0.409**	0.453	0.031
DIS	HRRFr	→	✓	0.810	0.786	0.054	0.007	0.000	0.057
		→	✗	0.639	0.500	0.251	0.032	0.019	0.052
		↔	✓	0.985	0.914	0.193	0.026	0.000	0.057
		↔	✗	0.982	0.912	0.291	0.045	0.003	0.056
	Mamba	→	✓	0.995	0.964	0.509	0.298	0.366	0.036
		→	✗	0.943	0.874	0.412	0.270	0.342	0.039
		↔	✓	**0.996**	0.967	0.609	0.401	0.356	0.035
		↔	✗	0.980	0.923	0.593	0.398	0.446	0.032
DEC	HRRFr	→	✓	0.991	0.950	0.327	0.050	0.058	0.048
		→	✗	0.986	0.939	0.182	0.023	0.002	0.055
		↔	✓	0.984	0.936	0.168	0.022	0.003	0.054
		↔	✗	0.982	0.941	0.282	0.044	0.000	0.055
	Mamba	→	✓	**0.996**	0.965	0.512	0.288	0.367	0.034
		→	✗	0.890	0.810	0.382	0.244	0.331	0.038
		↔	✓	0.994	0.960	0.608	0.402	0.346	0.034
		↔	✗	0.978	0.922	0.614	0.406	0.430	0.031
ALL	HRRFr	→	✓	0.993	0.955	0.268	0.035	0.065	0.047
		→	✗	0.995	0.968	0.062	0.009	0.027	0.049
		↔	✓	0.994	0.966	0.268	0.038	0.060	0.048
		↔	✗	0.975	0.923	0.237	0.033	0.131	0.043
	Mamba	→	✓	0.994	0.957	0.475	0.197	0.382	0.032
		→	✗	0.990	0.951	0.452	0.279	0.294	0.038
		↔	✓	**0.996**	0.968	0.584	0.346	**0.454**	**0.028**
		↔	✗	0.990	0.942	**0.633**	**0.409**	0.426	0.031
RAW	MalConv	↔	✗	**0.996**	0.970	0.600	0.407	0.322	0.033

TABLE II: Classification results, including malware detection, family classification, and behavioral tagging. The best value within each input group is highlighted; the best value across all inputs is boldfaced.

modeling could uncover more robust, general-purpose representations. Beyond the LMLM itself, the striking levels of redundancy we identified in malware corpora warrant an in-depth investigation to better understand the impact of this phenomenon on existing malware benchmarks. Also, to bridge the gap between human and machine, developing explainable AI techniques tailored to LMLMs remains an open area. Such techniques provide the transparency needed for security practitioners to trust model decisions and integrate them effectively into their threat analysis processes. Finally, adapting LMLMs for generative tasks would be intriguing.

VIII. Related Work

Due to its high performance in practice, many ML models for static malware analysis use handcrafted feature sets, such as the Drebin [79] (APK) and Ember (PE) [77] features. As DNNs have grown in popularity, we have observed a shift toward methods with reduced human involvement, such as classifiers based on control flow graphs like MalGraph [26] and MaMaDroid [28]. Principal among these are the raw-byte classifiers that detect malware by 'eating' entire files. MalConv [1], and its successors MalConvGCT [2] and CAG-MalConv [4], run inputs through 1-D convolutions to reduce the length of the input. This step reduces the richness of the

		Input				Architecture		Directedness		Pretrained	
		EXE	DIS	DEC	ALL	HRRFr	Mamba	→	↔	✓	✗
Detection	ROC ↑	0.970	0.916	0.975	**0.991**	0.948	0.979	0.942	0.985	0.975	0.952
	ACC ↑	0.840	0.855	0.928	**0.954**	0.854	0.934	0.867	0.921	0.906	0.882
Family	MCC ↑	0.348	0.364	0.384	0.372	0.200	**0.534**	0.311	0.423	0.375	0.359
	BAC ↑	0.191	0.185	0.185	0.168	0.028	**0.336**	0.149	0.215	0.181	0.184
Behavior	JAC ↑	0.186	0.191	0.192	0.230	0.023	**0.377**	0.183	0.217	0.200	0.200
	HAM ↓	0.045	0.046	0.044	0.039	0.053	**0.034**	0.044	0.042	0.043	0.044

TABLE III: Average of classification results between experiments with different experimental parameters. The best value within each category (Input, Architecture, Directedness, Pretraining) is highlighted; the best value across categories is boldfaced.

signal, as critical information is inevitably lost and inaccessible to deeper layers within the network. On the whole, this makes these architectures less capable of scaling by adding consecutive layers and also makes them less capable of performing the complex pretraining as used in LLMs. In contrast, our code-only representations extract the important parts of binaries as a preprocessing step, so our inputs are much smaller to begin with, and our tokenization strategy losslessly compresses sequences even further. This, in conjunction with the findings of our scaling analysis, allows us to employ advanced architectures with up to $60\times$ more non-embedding parameters than proposed by the original authors.

Orthogonal to how to represent malware for learning is the question of how to retain classification performance as malware evolves. Foundational works like TESSERACT [67] establish a strict framework for the evaluation of malware detection systems over time. Rahman et al. [78] found that existing *continual learning* techniques fall short in the malware domain, and in response, strategies have been proposed to keep malware systems up to date, such as active learning [63]. In addition to the natural evolution of malware as computer infrastructure changes, model designers need to contend with intentional obfuscation by malware creators. One approach centers around first detecting the presence and type of obfuscation a binary might have [80], and then removing said obfuscation before making a classification decision [81]. Another strategy is to simply use classifiers and feature sets that are more robust to obfuscation, e.g., by training models with obfuscated binaries [82]. Finally, the potential for carefully crafted adversarial malware [65], [83], [84] is a growing concern. It appears that all malware classifiers are susceptible to adversarial examples [5], so researchers are investing substantially in adversarial hardening techniques [6].

The broader area of binary analysis has also seen many advancements with the introduction of machine learning. Binary analysis often goes hand-in-hand with malware analysis, but serves other purposes as well, such as protecting intellectual property or finding bugs. Earlier works trained models in an end-to-end manner for a specific task, such as Gemini for binary code similarity [85], EKLAVYA for function type signature inference [86], and DEEPVSA for value-set analysis [87]. The success of task-agnostic pretraining in the NLP domain, particularly BERT [49], inspired generic models for binary code understanding like PalmTree [8] that could be finetuned for a variety of tasks. In contrast to our work, these methods are only applicable for building representations of small segments of code, such as functions, and it would not be computationally feasible to apply them to long-sequence malware analysis due to non-linear scaling properties. Active areas of research in this field include dealing with diverse instruction architecture sets [35] and automated binary lifting [88].

Finally, LLMs have recently shown promise within a variety of areas in security, for example, to improve performance at very hard binary analysis tasks like variable name recovery [89]. Within malware, some works have assessed LLMs' ability to analyze malware and perform categorization [90]. Considering the scale at which malware detection models need to be deployed, we argue that natural language LLMs are wasteful and impractical, as much of their knowledge has no bearing on malware understanding. We are the first to propose training large malware language models from scratch that can actually be utilized in a realistic malware analysis pipeline.

IX. Conclusion

In this work, we have taken an important step toward establishing large-scale representation learning in the malware domain. By examining every stage of the LMLM pipeline—**data**, **architecture**, **pretraining**, and **finetuning**—we have demonstrated a practical methodology for end-to-end learning from PE binaries at scale. Our experimental results underscore the promise of these approaches. Of these, our most critical novel finding is as follows: *pretraining models on unlabeled data using language modeling objectives improves downstream malware classification performance.* While we have only begun to scratch the surface of the language modeling paradigm in the malware context, this paper lays a solid groundwork upon which future research can build. Ultimately, by looking beyond raw bytes and embracing end-to-end learning scale, we move closer to a new generation of malware analysis.

Acknowledgments

This research was funded in part by the National Science Foundation under Grant no. 2422241. The authors acknowledge Research Computing at the Rochester Institute of Technology for providing computational resources and support.

Ethics Considerations

As generative models, LMLMs could be used to create new malware code. At the moment, the severity of this threat does not exceed that posed by existing code generation models, which have already been shown capable of generating malware C code. Nevertheless, the techniques explored in this work could be enhanced with the goal of generating novel malware variants in a completely end-to-end manner. LMLMs also give adversaries the opportunity to explore unprecedented attack vectors. For example, an adversary could drop an LMLM to a victim then remotely query it to create and execute a malware binary. Our current method and code are not easy to adapt to this end, so we believe that the benefits to the security community of making our code open source outweigh any risks. As future work on LMLMs advances, this tradeoff will need to be continually revisited.

References

[1] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *Workshop on Artificial Intelligence for Cyber Security (AICS)*, 2018.

[2] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean, "Classifying sequences of extreme length with constant memory applied to malware detection," in *Conf. on Artificial Intelligence (AAAI)*, 2021.

[3] M. M. Alam, E. Raff, S. Biderman, T. Oates, and J. Holt, "Recasting self-attention with holographic reduced representations," in *Int. Conf. on Machine Learning (ICML)*, 2023.

[4] X. Luo, H. Fan, L. Yin, S. Jia, K. Zhao, and H. Yang, "CAG-Malconv: A byte-level malware detection method with CBAM and Attention-GRU," *IEEE Trans. on Netw. and Service Management (TNSM)*, 2024.

[5] X. Ling, Z. Wu, B. Wang, W. Deng, J. Wu, S. Ji, T. Luo, and Y. Wu, "A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based Windows malware detection in the wild," in *USENIX Security Symp.*, 2024.

[6] K. Lucas, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif, "Training robust ML-based raw-binary malware detectors in hours, not months," in *ACM Conf. on Comput. and Commun. Security (CCS)*, 2024.

[7] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, "Training compute-optimal large language models," in *Conf. on Neural Inf. Process. Syst. (NeurIPS)*, 2022.

[8] X. Li, Y. Qu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *ACM Conf. on Comput. and Commun. Security (CCS)*, 2021.

[9] "OpenAI Models," platform.openai.com/docs/models.

[10] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," in *Conf. on Language Modeling (COLM)*, 2024.

[11] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv*, 2020.

[12] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long Range Arena: A benchmark for efficient Transformers," in *Int. Conf. on Learning Representations (ICLR)*, 2021.

[13] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *IEEE Int. Conf. on Comput. Vision (ICCV)*, 2017.

[14] C. G. Northcutt, A. Athalye, and J. Mueller, "Pervasive label errors in test sets destabilize machine learning benchmarks," in *Conf. on Neural Inf. Process. Syst. (NeurIPs) Datasets and Benchmarks Track*, 2021.

[15] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "PaLM: Scaling language modeling with pathways," *Journal of Machine Learning Research (JMLR)*, 2023.

[16] "Malware statistics and trends report," av-test.org/en/statistics/malware/.

[17] C. Liu, R. Saul, Y. Sun, E. Raff, M. Fuchs, T. S. Pantano, J. Holt, and K. Micinski, "Assemblage: Automatic binary dataset construction for machine learning," in *Conf. on Neural Inf. Process. Syst. (NeurIPs) Datasets and Benchmarks Track*, 2024.

[18] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "BODMAS: An open dataset for learning based temporal analysis of PE malware," in *Deep Learning Security and Privacy Workshop (DLSP)*, 2021.

[19] R. Harang and E. M. Rudd, "SOREL-20M: A large scale benchmark dataset for malicious PE detection," in *Conf. on Applied Machine Learning in Inf. Security (CAMLIS)*, 2021.

[20] "Detect-It-Easy," github.com/horsicq/Detect-It-Easy.

[21] "Ghidra," github.com/NationalSecurityAgency/ghidra.

[22] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *IEEE Symp. on Security and Privacy (SP)*, 2015.

[23] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Annual Comput. Security Appl. Conf. (ACSAC)*, 2007.

[24] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin' heat; limits of machine learning classifiers based on static analysis features," in *Netw. and Distributed Syst. Security Symp. (NDSS)*, 2020.

[25] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *IEEE/IFIP Int. Conf. on Dependable Syst. and Netw. (DSN)*, 2019.

[26] X. Ling, L. Wu, W. Deng, Z. Qu, J. Zhang, S. Zhang, T. Ma, B. Wang, C. Wu, and S. Ji, "MalGraph: Hierarchical graph neural networks for robust Windows malware detection," in *IEEE Int. Conf. on Comput. Commun. (INFOCOM)*, 2022.

[27] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *IEEE/ACM Conf. on Automated Software Engineering (ASE)*, 2019.

[28] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building markov chains of behavioral models," *IEEE Trans. on Privacy and Security (TOPS)*, 2019.

[29] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," in *Conf. on Neural Inf. Process. Syst. (NeurIPs)*, 2019.

[30] "LIEF," github.com/lief-project/LIEF.

[31] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *IEEE Symp. on Security and Privacy (SP)*, 2021.

[32] M. Zou, A. Khan, R. Wu, H. Gao, A. Bianchi, and D. J. Tian, "D-Helix: A generic decompiler testing framework using symbolic differentiation," in *USENIX Security Symp.*, 2024.

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Conf. on Neural Inf. Process. Syst. (NeurIPS)*, 2013.

[34] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Int. Conf. on Internet (ICONI)*, 2017.

[35] J. Wang, M. Sharp, C. Wu, Q. Zeng, and L. Luo, "Can a deep learning model for one architecture be used for others? Retargeted-architecture binary code analysis," in *USENIX Security Symp.*, 2023.

[36] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Assoc. for Computational Linguistics (ACL)*, 2016.

[37] G. Dagan, G. Synnaeve, and B. Roziere, "Getting the most out of your tokenizer for pre-training and domain adaptation," in *Int. Conf. on Machine Learning (ICML)*, 2024.

[38] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *Assoc. for Computational Linguistics (ACL)*, 2018.

[39] C. W. Schmidt, V. Reddy, H. Zhang, A. Alameddine, O. Uzan, Y. Pinter, and C. Tanner, "Tokenization is more than compression," in *Conf. on Empirical Methods in Natural Language Process. (EMNLP)*, 2024.

[40] E. M. Rudd, M. S. Rahman, and P. Tully, "Transformers for end-to-end infosec tasks: A feasibility study," in *Workshop on Rethinking Malware Analysis (WoRMA)*, 2022.

[41] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," in *Assoc. for Computational Linguistics (ACL)*, 2022.

[42] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv*, 2023.

[43] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRo: Memory optimizations toward training trillion parameter models," in *Int. Conf. for High Performance Comput., Netw., Storage and Analysis (SC)*, 2020.

[44] J. Ba and D. Kingma, "Adam: A method for stochastic optimization," in *Int. Conf. on Learning Representations (ICLR)*, 2015.

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Conf. on Neural Inf. Process. Syst. (NeurIPS)*, 2017.

[46] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient Transformers: A survey," *ACM Computing Surveys (CSUR)*, 2022.

[47] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "RoFormer: Enhanced Transformer with rotary position embedding," *Neurocomputing*, 2024.

[48] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large Transformer models," in *Conf. on Machine Learning and Syst. (MLSys)*, 2023.

[49] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional Transformers for language understanding," in *North American Assoc. for Computational Linguistics (NAACL)*, 2019.

[50] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling vision Transformers," in *IEEE/CVF Conf. on Comput. Vision and Pattern Recognition (CVPR)*, 2022.

[51] Q. Yao, C.-H. H. Yang, R. Jiang, Y. Liang, M. Jin, and S. Pan, "Towards neural scaling laws for time series foundation models," *arXiv*, 2024.

[52] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv*, 2016.

[53] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised searning of language representations," in *Int. Conf. on Learning Representations (ICLR)*, 2020.

[54] A. Radford, "Improving language understanding by generative pre-training," *OpenAI Blog*, 2018.

[55] T. A. Plate, "Holographic reduced representations," *IEEE Trans. on Neural Netw.*, 1995.

[56] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Assoc. for Computational Linguistics (ACL)*, 2020.

[57] R. J. Joyce, D. Amlani, C. Nicholas, and E. Raff, "MOTIF: A malware reference dataset with ground truth family labels," *Computers and Security (C&S)*, 2023.

[58] X. Wu, W. Guo, J. Yan, B. Coskun, and X. Xing, "From grim reality to practical solution: Malware classification in real-world noise," in *IEEE Symp. on Security and Privacy (SP)*, 2023.

[59] A. Wettig, T. Gao, Z. Zhong, and D. Chen, "Should you mask 15% in masked language modeling?" in *European Assoc. for Computational Linguistics (EACL)*, 2023.

[60] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring," in *Assoc. for Computational Linguistics (ACL)*, 2020.

[61] C. Tao, Q. Liu, L. Dou, N. Muennighoff, Z. Wan, P. Luo, M. Lin, and N. Wong, "Scaling laws with vocabulary: Larger models deserve larger vocabularies," in *Conf. on Neural Inf. Process. Syst. (NeurIPS)*, 2024.

[62] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking Android apps via a hardware-assisted approach," in *IEEE Symp. on Security and Privacy (SP)*, 2021.

[63] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for Android malware detection," in *USENIX Security Symp.*, 2023.

[64] T. Zheng, Q. Hou, X. Chen, H. Ren, M. Li, H. Li, and C. Shen, "Gupacker: Generalized unpacking framework for Android malware," *IEEE Trans. on Inf. Forensics and Security (TIFS)*, 2025.

[65] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ML attacks in the problem space," in *IEEE Symp. on Security and Privacy (SP)*, 2020.

[66] D. Gibert, N. Totosis, C. Patsakis, G. Zizzo, and Q. Le, "Assessing the impact of packing on machine learning-based malware detection and classification systems," *Computers and Security (C&S)*, 2025.

[67] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *USENIX Security Symp.*, 2019.

[68] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge, "Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance," in *ACM Conf. on Comput. and Commun. Security (CCS)*, 2023.

[69] "Virus Total," virustotal.com.

[70] S. Sebastián and J. Caballero, "AVClass2: Massive malware tag extraction from AV labels," in *Annual Comput. Security Appl. Conf. (ACSAC)*, 2020.

[71] Y. Cui, M. Jia, T.-Y. Lin, Y. Song, and S. Belongie, "Class-balanced loss based on effective number of samples," in *IEEE/CVF Conf. on Comput. Vision and Pattern Recognition (CVPR)*, 2019.

[72] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *USENIX Security Symp.*, 2022.

[73] R. J. Joyce, E. Raff, C. Nicholas, and J. Holt, "MalDICT: Benchmark datasets on malware behaviors, platforms, exploitation, and packers," in *Conf. on Applied Machine Learning for Inf. Security (CAMLIS)*, 2023.

[74] T. Wu, Q. Huang, Z. Liu, Y. Wang, and D. Lin, "Distribution-balanced loss for multi-label classification in long-tailed datasets," in *European Conf. on Comput. Vision (ECCV)*, 2020.

[75] T.-Y. Ross and G. Dollár, "Focal loss for dense object detection," in *IEEE Conf. on Comput. Vision and Pattern Recognition (CVPR)*, 2017.

[76] Z. Wang, X. Ning, and M. B. Blaschko, "Jaccard metric losses: Optimizing the jaccard index with soft labels," in *Conf. on Neural Inf. Process. Syst. (NeurIPs)*, 2023.

[77] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static PE malware machine learning models," *arXiv*, 2018.

[78] M. S. Rahman, S. Coull, and M. Wright, "On the limitations of continual learning for malware classification," in *Conf. on Lifelong Learning Agents (CoLLAs)*, 2022.

[79] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Netw. and Distributed Syst. Security Symp. (NDSS)*, 2014.

[80] S. Li, J. Ming, P. Qiu, Q. Chen, L. Liu, H. Bao, Q. Wang, and C. Jia, "PackGenome: Automatically generating robust YARA rules for accurate malware packer detection," in *ACM Conf. on Comput. and Commun. Security (CCS)*, 2023.

[81] B. Cheng, E. A. Leal, H. Zhang, and J. Ming, "On the feasibility of malware unpacking via hardware-assisted loop profiling," in *USENIX Security Symp.*, 2023.

[82] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "DANdroid: A multi-view discriminative adversarial network for obfuscated Android malware detection," in *ACM Conf. on Data and Application Security and Privacy (CODASPY)*, 2020.

[83] H. Berger, C. Hajaj, E. Mariconti, and A. Dvir, "Crystal ball: From innovative attacks to attack effectiveness classifier," *IEEE Access*, 2021.

[84] H. Berger, A. Dvir, E. Mariconti, and C. Hajaj, "Breaking the structure of MaMaDroid," *Expert Syst. with Appl. (ESWA)*, 2023.

[85] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Conf. on Comput. and Commun. Security*, 2017.

[86] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *USENIX Security Symp.*, 2017.

[87] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in *USENIX Security Symp.*, 2019.

[88] S. Yu, Y. Qu, X. Hu, and H. Yin, "DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *USENIX Security Symp.*, 2022.

[89] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "ReSym: Harnessing LLMs to recover variable and data structure symbols from stripped binaries," in *ACM Conf. on Comput. and Commun. Security (CCS)*, 2024.

[90] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro, "LAMD: Context-driven Android malware detection and classification with LLMs," *arXiv*, 2025.

[91] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Int. Conf. on Machine Learning (ICML)*, 2017.

APPENDIX A
SYMBOL TABLE

Sym.	Sec.	Definition
NLP	I	Natural Language Processing
DNN	I	Deep Neural Network
LLM	I	Large Language Model
LMLM	I	Large Malware Language Model
SSM	IV-A	[Neural] State Space Model
RNN	IV-A	Recurrent Neural Network
CLM	V-A	Causal Language Modeling
MLM	V-A	Masked Language Modeling
RAW	II-B	Raw bytes of a binary
EXE	II-B	Bytes extracted from the .text section
DIS	II-B	Stream of disassembled instructions
DEC	II-B	Decompiled, C-like code
ALL	VI-B	Combination of EXE, DIS, and DEC
BPE	II-C	Byte pair encoding tokenization
UNI	II-C	Unigram language model tokenization
BYT	II-C	No tokenization (raw-bytes or ASCII)
T	IV-A	Length of a malware sequence
H	IV-A	Hidden size of a neural network
L	IV-A	Number of layers in a neural network
I	IV-B	Intermediary size of linear projections
S	IV-B	Size of Mamba's state space model
$\mathcal{V}$, V	IV-A	Vocabulary of discrete tokens and its size
Φ, ϕ	IV-A	Sequence processing network and blocks
$\mathbf{x}, x$	IV-A	Sequence of (and one) token(s) for sample
$\mathbf{z}, z$	IV-A	Sequence of (and one) latent vector(s)
$\mathcal{E}$	IV-A	Embedding table for $\mathcal{V}$
k	IV-C	Transformer key vectors
q	IV-C	Transformer query vectors
v	IV-C	Transformer value vectors
β	IV-C	HRRFormer superposition
$\hat{v}$	IV-C	HRRFormer value approximations
$\circledast$	IV-C	HRR binding operation
$\dagger$	IV-C	HRR unbinding operation
Θ	V-A	Parameters of model
$P_\Theta(\cdot)$	V-A	Pseudo class probability given by Θ
$P_\mathcal{V}(\cdot)$	V-A	Unigram probability of token in $\mathcal{V}$
$\mathbf{m}$	V-A	Boolean mask for pretraining
$\mathbf{w}, w$	V-A	Sequence of (and one) token(s)
$\mathbf{c}$	V-A	Context, i.e., a subset of $\mathbf{w}$
PPL	V-A	Perplexity
NPPL	V-A	Normalized Perplexity
ROC	VI-C	Area under the receiver operator curve
ACC	VI-C	Accuracy
MCC	VI-D	Matthews correlation coefficient
BAC	VI-D	Balanced accuracy
JAC	VI-E	Jaccard index
HAM	VI-E	Hamming loss
λ	V-B	AdamW optimizer weight decay
γ	V-B	AdamW optimizer learning rate
β	VI-D	Sample reweighting's weighting factor
α	VI-E	Focal loss's weighting factor
γ	VI-E	Focal loss's modulating factor
$\rightarrow$	IV-C	Unidirectional architecture
$\leftrightarrow$	IV-C	Bidirectional architecture
✓	VI	Model was pretrained
✗	VI	Model wasn't pretrained

TABLE IV: Symbols and abbreviations used in this work.

APPENDIX B
COMPUTATIONAL COSTS

Automated malware analysis needs to take place in a timely manner. Exactly how timely is necessary can vary by organization and by use case. Compared to dynamic analysis, statically extracted features are almost always going to be more efficient to attain. Our preprocessing/tokenization is performed with LIEF, Ghidra, tokenizers, and Cython kernels. To measure the cost of the proposed preprocessing routines, we run 2000 randomly selected malware samples from the Sorel corpus and 2000 randomly selected goodware samples from the Assemblage corpus through our preprocessing and tokenization pipeline on a server with twenty SMT-enabled physical cores @2.40GHz and 256GB RAM. We compare the results with the time required to extract Ember features [77]. The average time (in seconds) taken per sample is recorded in Table V. We provide similar statistics about the time each network takes to process a single sample, averaged over the entire pretraining run, and the number of GPU hours required to train the models for each task in Table VI and Table VII, respectively. All models were trained on NVIDIA PCIe A100s using data parallelism implemented with transformers and PyTorch. The amount of memory and processors allocated through SLURM varies per task, but did not exceed 16 SMT-enabled physical cores @2.70GHz and 360GB RAM.

		EXE	DIS	DEC	Ember
Preprocess	Malware	0.002	8.01	14.0	0.014
	Benign	0.001	2.57	5.44	0.007
Tokenize	Malware	0.004	0.002	0.002	—
	Benign	0.002	0.002	0.002	—

TABLE V: Time taken (in seconds) to prepare a single binary using the preprocessing and tokenization routines.

Architecture	**Directedness**	**Train**	**Inference**
HRRFr	$\rightarrow$	1.32	0.357
	$\leftrightarrow$	0.852	0.260
Mamba	$\rightarrow$	1.21	0.456
	$\leftrightarrow$	1.22	0.457
MalConv	$\leftrightarrow$	0.087	0.068

TABLE VI: Time taken (in seconds) for each architecture to process a single sample when training and during inference.

Architecture	**Direct.**	CLM	MLM	DET	FAM	BEH
HRRFr	$\rightarrow$	558	—	45.1	96.4	25.3
	$\leftrightarrow$	—	230	29.1	63.0	15.9
Mamba	$\rightarrow$	376	—	41.4	90.5	24.4
	$\leftrightarrow$	—	382	41.7	91.0	24.4
MalConv	$\leftrightarrow$	—	—	2.81	5.27	1.39

TABLE VII: GPU hours (#GPUs $\times$ #hours) for each architecture to be trained for each task (excludes model validation).

APPENDIX C
TOKENIZATION COMPRESSION

In Table VIII, we display the lengths and compression ratios of each configuration. On each input, we measure the compression ratio compared to the byte-level baseline within that input (Comp-1). Across all inputs, we measure the compression ratio compared to the byte-level approach utilizing the entire binary (Comp-2).

Input	Vec.	Vocab	Length (↓)	Comp-1 (↑)	Comp-2 (↑)
EXE	BYT	256	57.3	—	18.2
	BPE	1024	36.4	1.58	28.7
		4096	32.2	1.78	32.5
		16384	27.4	2.09	38.1
	UNI	1024	38.8	1.48	26.9
		4096	34.2	1.68	30.6
		16384	28.8	1.99	36.2
DIS	BYT	256	50.5	—	20.7
	BPE	1024	8.90	5.70	118
		4096	7.50	6.75	140.
		16384	**6.60**	**7.66**	**159**
	UNI	1024	9.70	5.23	108
		4096	8.60	5.86	121
		16384	7.80	6.48	134
DEC	BYT	256	160.	—	6.52
	BPE	1024	42.8	3.74	24.4
		4096	29.8	5.38	35.1
		16384	23.3	6.87	44.8
	UNI	1024	42.8	3.74	24.4
		4096	30.9	5.19	33.8
		16384	24.5	6.54	42.7
RAW	BYT	256	1040	—	—
	BPE	1024	708	1.48	—
		4096	612	1.71	—
		16384	475	2.20	—
	UNI	1024	784	1.33	—
		4096	597	1.75	—
		16384	479	2.18	—

TABLE VIII: Impact of vocabulary on the sequence length. 'Length' shows the median length in units of 10^3 tokens. The best value within each input representation is highlighted; the best value across all inputs is boldfaced.

APPENDIX D
DATASET DETAILS

In Figure 7, we display the distribution of malware families in our preprocessed datasets, after removing packed and redundant samples. We categorize the families into broader groups using the 'class' field returned by AVClass2 [70]. We display similar information for the distribution of malware behaviors defined by ClarAVy [73]. Here, we categorize behaviors into one of the malware 'objectives' defined under the MITRE Malware Behavior Catalog (MBC)[2]. Note that these distributions represent the corpus as a whole, but not necessarily each split in the finetuning datasets.

[2]https://github.com/MBCProject/mbc-markdown

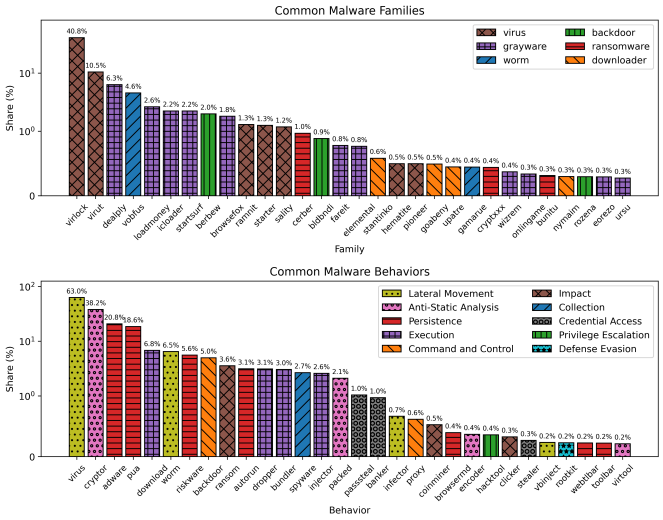


Fig. 7: Thirty most frequent malware families and behaviors. Note the multilabel behavioral tags will not add to 100%.

APPENDIX E
VOCABULARY ABLATIONS

To explore the impact of vocabulary on the model's ability to learn, we conduct an ablation study using BPE and UNI tokenizers with vocabularies of 1024, 4096, and 16384 tokens for language modeling on EXE inputs with Mamba. For computational feasibility, we use a smaller network with $H = 192$ and $L = 32/16$, which contains approximately 8M non-embedding parameters (we'll refer to this as a 'mini' LMLM in the following appendices). We train each network on 400K samples. Since we're dealing with different vocabularies, we only report the normalized perplexity (lower is better). The results are displayed in Figure 8. Here we see that i) models trained with BPE vocabularies achieve lower NPPL than those trained with UNI vocabularies; ii) models trained with larger vocabulary sizes tend to achieve lower NPPL than those trained with smaller vocabularies; and iii) these smaller models achieve significantly worse performance than the large models trained in our main experiments (Figure 5).

APPENDIX F
DETECTION ABLATIONS

As previously discussed, there is no widely agreed upon value representing the ratio of PE malware to goodware in the wild. In this section, we perform an ablation using ratios of 10%, 30%, 50%, 70%, and 90% malware for malware detection on EXE inputs with 'mini' Mamba. We train each network for 2048 steps with a batch size of 64. Since the data is imbalanced, we report the F-1 score, which is defined as the harmonic mean of precision and recall. The results are displayed in Figure 9. Here, we see that pretraining is especially beneficial in scenarios where goodware is more prevalent than malware.

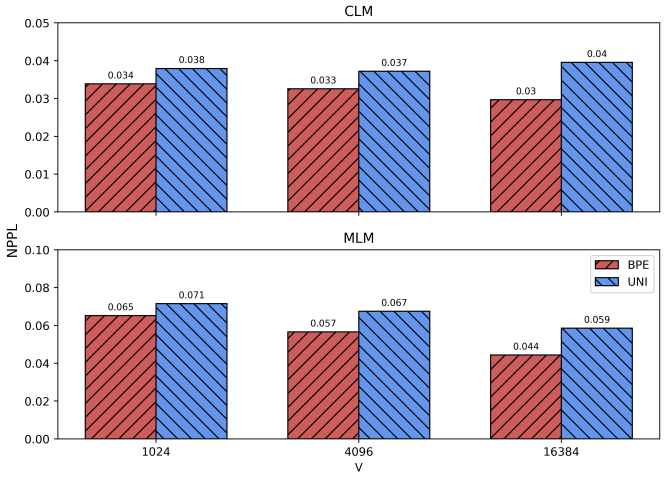


Fig. 8: Performance of causal and masked language models trained to model malware with different vocabularies.

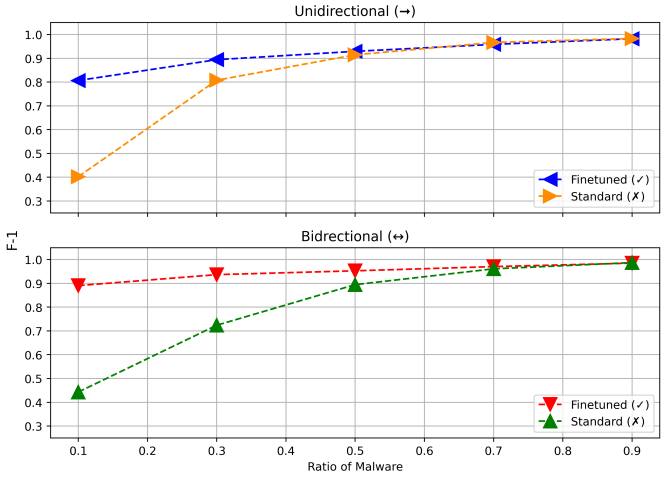


Fig. 9: Performance of binary malware classifiers under different assumed ratios of malware-goodware.

Appendix G
Packing Ablations

As previously discussed, to quantify models' ability to learn from byte-level, disassembly, and decompiled code, we filtered encrypted and compressed samples from our training corpora since packed binaries cannot be lifted. To demonstrate that our approach is applicable for malware detection post-unpacking, we perform an ablation with not-packed, packed, and unpacked samples for malware detection on EXE inputs with 'mini' bidirectional Mamba. In these experiments, the validation set contains an equal distribution of samples that are either not packed, packed with UPX, or packed and then unpacked with UPX, while the training sets are configured to exclude some of these types of samples. The results in Table IX indicate that EXE preprocessing with Mamba is robust to packed samples when included in the training corpus, but that unpacking is key to unlocking the full potential of LMLMs.

Train			Test		
No-Pack	**Pack**	**Un-Pack**	**No-Pack**	**Pack**	**Un-Pack**
✓	✗	✗	0.893	0.512	0.889
✗	✓	✗	0.634	0.939	0.647
✗	✗	✓	0.904	0.634	0.832
✓	✓	✗	0.855	0.919	0.844
✓	✗	✓	0.879	0.614	0.898
✗	✓	✓	0.803	0.909	0.853
✓	✓	✓	0.857	0.913	0.864

TABLE IX: Accuracy of malware detector when trained with and without packed and unpacked samples. A green check-mark (✓) indicates that data is included in the training set, while a red x-mark (✗) indicates that data is excluded from the training set. The right-hand portion of the table details the accuracy over each portion of the test set.

Appendix H
Ensemble Analysis

To provide further insight into the mechanisms of the ensemble model, we conduct a post-hoc explanation analysis on 1000 samples from the test set of the detection task. We use Integrated Gradients [91] to assess the impact of the EXE, DIS, and DEC token embeddings toward the desired classification result. We normalize the attribution scores for each sample and compute the average contribution of each input type. The results are displayed in Figure 10. Here we see that i) different architectures consider different inputs more salient; ii) pretraining changes how the same architecture arrives at a classification decision; and iii) aside from the unidirectional model trained from scratch, the input type most useful for each architecture is more or less consistent across different tasks.

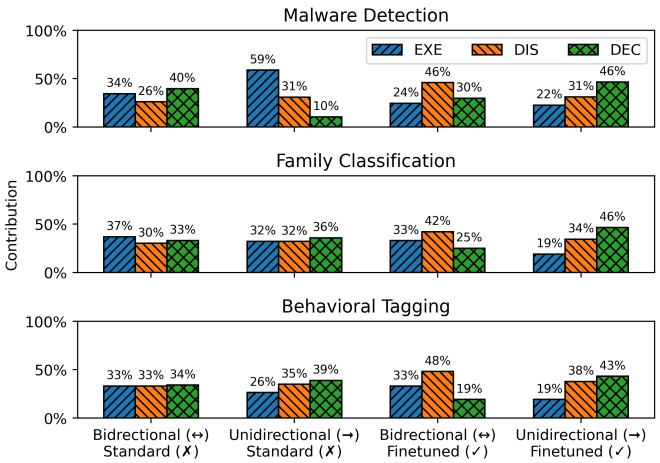


Fig. 10: Average contribution of each input type to the true classification label for the ensemble classifiers.

APPENDIX I
ARTIFACT APPENDIX

To promote open-source science, we make *available* our code along with a collection of experiments demonstrating its *functionality*. At this moment, it is not feasible to make our results completely *reproducible* due to the extensive computational requirements to do so. Due to licensing, we cannot release our data, but it is publicly available elsewhere.

A. Description & Requirements

1) How to access: Our artifact is accessible via a GitHub repository[3] and archived permanently using Zenodo[4].

2) Hardware dependencies: A functional demonstration requiring no specialized hardware is available. To reproduce our work, one would need access to a large server with multiple GPUs, multiple CPUs, a few hundred GBs RAM, and several TBs persistent storage.

3) Software dependencies: A functional demonstration requiring only Docker (containerization) and Zstandard (decompression) is available. In short, our core dependencies revolve around the CUDA, PyTorch and Huggingface ecosystems. A complete list of software packages and installation instructions using Anaconda is available on our repository. A Linux machine is recommended.

4) Benchmarks: Our work used data from the Assemblage, Bodmas, and Sorel corpora along with binaries from a Windows installation. Our functional demonstration includes a small sample of preprocessed samples derived from these datasets. Our code can be used with any PE binaries.

B. Artifact Installation & Configuration

To run the functional demonstration, navigate to the repository. Detailed instructions are available in the `README` file. Download the docker image from the Google Drive folder or Zenodo and setup the Docker container. Run the preliminary processes (building caches, training tokenizers, generating experiment configurations) before training and evaluating deep learning models.

C. Experiment Workflow

The workflow is embedded in the Evaluation section.

D. Major Claims

Below are claims to be substantiated by our artifact:

- ⟨**C$_1$**⟩ We propose and compare three code-only representations of PE malware for malware classification.
- ⟨**C$_2$**⟩ We use subword tokenization to reduce the length of malware sequences without loss of information.
- ⟨**C$_3$**⟩ We discover and correct previously unrealized sample redundancy in PE malware datasets.
- ⟨**C$_5$**⟩ We pretrain a suite of malware language models on causal and masked language modeling objectives.
- ⟨**C$_6$**⟩ We conduct classification experiments including detection, family classification, and behavioral tagging.

[3]https://github.com/lkurlandski/RawByteClf

[4]DOI: 10.5281/zenodo.17047102

E. Evaluation

Here we describe the evaluation to be performed using our demo. As previously stated, it is not feasible to reproduce our results in a short period of time. The objective of this evaluation to demonstrate that our code base is capable of running our experiments and producing our results.

1) Prerequisites: **[10 minutes]** To run the functional demonstration, navigate to the repository. The `README` file contains detailed instructions to run each step of the functional demonstration. The only software required is Docker and the Zstandard. Docker commands require root-level (`sudo`) access. If these requirements are met, the demo can be run.

2) Setup: **[5 minutes]** First, download each shard of the Docker image (`demo.tar.zst.part-*`) from the Google Drive folder linked in the `README` or from Zenodo. Concatenate them together. Since it contains a small preprocessed dataset, the image is relatively large (9GB decompressed). As a consequence, the default Docker mount in `/var/lib/docker/` may not be large enough to run the image. To attain additional space, we provide instructions to either cleanup existing Docker artifacts or switch Docker's mount to a device with more storage capacity (20GB required). After the image is successfully created, `sudo docker run --rm demo:latest` should print 'Hello World!'.

3) Preparation: **[5 minutes]** Next, start the Docker daemon. All subsequent `sudo docker exec` commands will execute inside of the currently running daemon. Before conducting experiments, allocate samples to train and test splits for each experiment by running the `gencaches.sh` script. Also, prepare a batch of `.sh` files to drive deep learning experiments later on using the `create.py` script.

4) Exploration: **[5 minutes]** ⟨**C$_1$**⟩ At any stage, data within the container can be copied to a local destination and inspected. At this moment, the only items of interest are the preprocessed samples, which are located in `/home/appuser/app/data/`. Here the directories `ass/`, `bod/`, `sor/`, and `win/` contain data from the Assemblage, Bodmas, Sorel, and Windows corpora, respectively. The subdirectories `nop/`, `raw/`, `dis/`, and `dec/` contain full binaries (RAW), bytes from executable sections (EXE), disassembly (DIS), and decompiled code (DEC), respectively[5]. To explore the preprocessed data, copy archives locally, extract samples, and inspect their contents with a text/hex editor.

5) Tokenization: **[15 minutes]** ⟨**C$_2$**⟩ Next, train tokenizers to tokenize each type of input with the `tokenize.sh` script. The vocabularies are saved in plaintext JSON files in `/home/appuser/app/output/tokenizers/`. Copy them locally and view the mapping between tokens and token ids. Note that each raw byte in the EXE input is first mapped to a UTF-8 symbol before tokenizing, so the tokens here are not recognizable as a subsequence of bytes. The vocabularies for DIS and DEC will contain common, recognizable instructions and C-code constructs.

[5]The codebase uses 'nop' to refer to RAW and 'raw' to refer to EXE.

6) Redundancy: **[5 minutes]** ⟨C$_3$⟩ For each dataset directory (`ass/`, `bod/`, `sor/`, and `win/`) and each representation subdirectory (`raw/`, `dis/`, and `dec/`), in `/home/appuser/app/data/`, there exists a `digests.json` file that maps the digest of the source PE binary to the digest of its corresponding EXE, DIS, or DEC representation. For each `digests.json` file, the `redundancy.py` script counts the number of unique binaries and compares this to the number of unique inputs within a specific representation. The results printed indicate that much of the malware corpora consist of samples with identical code whereas most of the goodware samples are more unique.

7) Pretraining: **[10 minutes]** ⟨C$_5$⟩ The `create.py` script creates 87 `.sh` files in `/home/appuser/app/demo/sbatch/`. Each file, when run, trains and evaluates a distinct model for a distinct learning task. These scripts contain 'toy' configurations (smaller models, less data, shorter sequence lengths, and only cursory training time) but are representative of the main body of experiments conducted in our paper. The `README` contains instruction to pretrain three malware masked/causal language HRRFormer/Mamba models on EXE, DIS, and DEC inputs. A total of twelve different configurations can be run by running the relevant bash script.

8) Classification: **[10 minutes]** ⟨C$_6$⟩ Following pretraining, a pretrained LMLM can be finetuned for a practical malware analysis task, such as malware detection, family classification, or behavioral tagging. Each pretrained model from the previous experiments can be finetuned for any of these tasks. The same architecture can be trained from scratch — without having been pretrained — and their performance compared.

9) Cleanup: When finished, terminate the Docker deamon and cleanup its state. Remove data copied locally.