StackBERT: Machine Learning Assisted Static Stack Frame Size Recovery on Stripped and Optimized Binaries

Chinmay Deshpande
cddespha@uci.edu
University of California, Irvine
Irvine, CA, USA

David Gens
dgens@uci.edu
University of California, Irvine
Irvine, CA, USA

Michael Franz
franz@uci.edu
University of California, Irvine
Irvine, CA, USA

ABSTRACT

The call stack represents one of the core abstractions that compiler-generated programs leverage to organize binary execution at runtime. For many use cases reasoning about stack accesses of binary functions is crucial: security-sensitive applications may require patching even after deployment, and binary instrumentation, rewriting, and lifting all necessitate detailed knowledge about the function frame layout of the affected program. As no comprehensive solution to the stack symbolization problem exists to date, existing approaches have to resort to workarounds like emulated stack environments, resulting in increased runtime overheads.

In this paper we present StackBERT, a framework to statically reason about and reliably recover stack frame information of binary functions in stripped and highly optimized programs. The core idea behind our approach is to formulate binary analysis as a self-supervised learning problem by automatically generating ground truth data from a large corpus of open-source programs. We train a state-of-the-art Transformer model with self-attention and finetune for stack frame size prediction. We show that our finetuned model yields highly accurate estimates of a binary function's stack size from its function body alone across different instruction-set architectures, compiler toolchains, and optimization levels. We successfully verify the static estimates against runtime data through dynamic executions of standard benchmarks and additional studies, demonstrating that StackBERT's predictions generalize to 93.44% of stripped and highly optimized test binaries not seen during training. We envision these results to be useful for improving binary rewriting and lifting approaches in the future.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Software maintenance tools**; • **Computing methodologies** → **Semi-supervised learning settings**.

KEYWORDS

binary lifting, recompilation, machine learning, stack symbolization

ACM Reference Format:
Chinmay Deshpande, David Gens, and Michael Franz. 2021. StackBERT: Machine Learning Assisted Static Stack Frame Size Recovery on Stripped and Optimized Binaries. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security (AISec '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3474369.3486865

AISec '21, November 15, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8657-9/21/11.
https://doi.org/10.1145/3474369.3486865

1 INTRODUCTION

Binary program analysis raises fundamental challenges [20]. One of the reasons for this is that a lot of semantic and contextual information about the program such as precise control flows, types, syntactic idioms, and symbols is present at the source level, but missing at the binary level. In theory, compilers could support parsing and operating on binaries to provide a continued, integrated approach to "binops". In practice, compiling software from source level down to the binary format accepted by contemporary computing hardware is a one-way street.

Nonetheless, in many cases working with binary programs represents an operational necessity, for instance, because recompilation from source is not desirable, feasible, or legal [24]. Even for open-source software binary rewriting may become necessary, e.g., when a system requires live patching. For this reason, a number of binary analysis and instrumentation approaches were proposed over time to systematically attack the fundamental challenges associated with binary analysis. Because the underlying problems can be difficult to solve entirely, many frameworks resolve to heuristics and relaxing assumptions to nail down indirect control flows, infer higher-level type information, and lift binary code to compiler-level intermediate representations [2, 13, 15].

One key aspect in binary patching is understanding the stack layouts of the patched functions [21]. While binary format standardization and debugging information can help in documenting binary stack operations to some extent, they may not always be available, sufficiently precise, or even correct [6, 42]—particularly in the context of heavy compiler optimizations. This is why state-of-the-art approaches currently resort to emulated stack environments for lifted binaries [2, 15], and live-patching of binary functions currently requires human-in-the-loop operation. To the best of our knowledge *stack symbolization* currently remains as a difficult, open problem in the literature and as a result there currently exists no practical approach to statically reason about stack accesses of binary programs for the purpose of patching or recompilation.

Owing to the fundamental difficulties in traditional approaches, recent years have seen a rise in the application of Machine Learning (ML) based approaches for binary analysis tasks. These approaches benefit through learning from a wide range of binaries

compiled for different architectures and across various optimization levels. They usually involve extracting features which convey high-level semantics of the underlying task to the model being trained. More recently, there have been attempts in treating assembly code as natural language and leveraging advances in natural-language processing (NLP) to solve analogous problems in binary analysis. However, existing approaches either focus on function identification [1, 5, 41, 45], feature extraction for decompilation [16, 17, 19, 36, 37], or type inference [14]. We note that inferring low-level information such as the stack layout of binary functions—which binary patching, recompilation, and lifting all require—is not a focus of any of these works.

Contributions. In this paper, we present StackBERT, a novel binary analysis approach that successfully tackles stack symbolization tasks in compiler-generated binaries in an instruction-set architecture (ISA) agnostic manner. Our core idea is to formulate stack frame recovery as a self-supervised learning problem by automatically generating labeled training data from open-source software using popular compiler toolchains. Our framework consists of two main components: (1) a deep neural network using the popular Transformer architecture with self-attention [29, 44] for prediction tasks in binary analysis, and (2) a set of tools to automatically extract and generate ground truth labels from compiler-generated binaries, as well as verifying model predictions for key stack symbolization tasks using runtime information.

In our detailed evaluation we demonstrate that this allows us to successfully solve stack symbolization tasks, such as frame size recovery for stripped and highly optimized binaries of real-world programs and standard benchmarks that were not seen during training. We also validate the stack symbolization predictions from the learned model against runtime data, further demonstrating that StackBERT is able to successfully solve stack symbolization tasks for 93.44% of unseen binaries in our tests. Given this level of reliability—which remains robust across optimization levels, compilers, and instruction-set architectures—we anticipate these results to be useful for moving towards native stack support in binary lifting, rewriting, and patching in the future.

In summary, we make the following contributions:

- We present StackBERT, a novel and architecture-agnostic approach to binary analysis that uses masked byte prediction to pretrain a Transformer model in a self-supervised manner. We demonstrate the efficacy of our trained models with respect to stack frame size recovery from the binary function body alone.
- We extensively test and evaluate our prototype implementation using two popular and widely used compiler suites, high optimization levels, and across different instruction set architectures, demonstrating 93.44% validation accuracy on standard benchmarks that were not seen during training.
- We present a new training set and tools to automatically extract features and labels from open-source software, which we release as an open-source implementation alongside our model and baseline implementations. [1]

[1] https://www.github.com/securesystemslab/stackbert

2 MOTIVATING EXAMPLES

Although stack symbolization of binary programs would enable many possible applications in binary rewriting, lifting, and recompilation, it currently remains as a largely open problem. There are several reasons for this: first, binary programs in principle are free not to use the system's call stack and arbitrary binaries that were produced using handwritten assembly, self-modifying code, or code virtualization obfuscation might explicitly opt not to. Second, even for binaries that are generated by standard compilers—and which consequently should adhere to the system's Application Binary Interface (ABI)—symbolizing stack accesses is uniquely challenging [10]. In fact, it can be nearly impossible for human developers to make sense of a program's stack management just by looking at stack traces post mortem: the Linux kernel developer community had to develop a dedicated stack metadata validator [38] and unwinder [22] to deal with mounting problems of garbled and unintelligible stack traces in bug reports and crash dumps.

To illustrate some of these challenges concretely let us consider a simple source-code example in Listing 1. It contains two function definitions as well as a global integer variable definition. Since `main` takes parameters and also defines local variables, one might expect a dedicated function frame. Moreover, as the function called by `main` is annotated with the `__inline__` intrinsic, one might also expect its stack frame to be part of `main`'s stack frame, given this particular source code. However, if we look at the disassembly of the binary in Listing 2 that is generated by the two big compiler suites GCC (in version 11.1.0) and LLVM (in version 13.0.0) with only minor variations under modest optimization levels (i.e., -O2 and -ansi), we discover that neither of these assumptions hold.

```
1  static volatile int n;
2  static __inline__ int inlineme(int * i) {
3      int foo[123] = {};
4      putchar(foo);
5      if (*i < 123)
6          return *i;
7  lt500:;
8      char x[n];
9      putchar(x);
10     if (n++ < 500)
11         goto lt500;
12     return -1;
13 }
14 int main (int argc, char *argv[]) {
15     int *b[10];
16     n = argc;
17     b[5] = (int *)&n;
18     return inlineme(b[5]);
19 }
```

Listing 1: Example C program (compiles with `-O2 -ansi`).

The reason is that both compilers determine the stacklayout of function `inlineme` to be incompatible with the stack layout of `main`, and hence, cannot inline the function. They do however determine `inlineme` to be tailcall optimizable and directly forward the local variable definitions through `main` via constant propagation. The result is that `inlineme` will be defined as a function symbol in the binary's symbol table despite being marked inline and never explicitly being called anywhere (only jumped to). Consequently, a crashdump of this program's execution if called with

more than 121 command line arguments (e.g., with "`seq 122 | xargs ./example`") will not show the `main` function being called. The exact same issue also arises during debugging. If this program required live patching in a production system, the call stack would not serve as a reliable source of its execution state. [2]

```
1  inlineme:
2          push    rbp
3          mov     rbp, rsp
4          push    rbx
5          lea     rdi, [rbp-512]
6          sub     rsp, 504
7          call    putchar
8          mov     eax, DWORD PTR n[rip]
9          cmp     eax, 122
10         jg      .L3
11 .L1:
12         mov     rbx, QWORD PTR [rbp-8]
13         leave
14         ret
15 .L7:
16         mov     rsp, rbx
17 .L3:
18         movsx   rax, DWORD PTR n[rip]
19         mov     rbx, rsp
20         add     rax, 15
21         and     rax, -16
22         sub     rsp, rax
23         mov     rdi, rsp
24         call    putchar
25         mov     eax, DWORD PTR n[rip]
26         lea     edx, [rax+1]
27         mov     DWORD PTR n[rip], edx
28         cmp     eax, 499
29         jle     .L7
30         mov     rsp, rbx
31         or      eax, -1
32         jmp     .L1
33 main:
34         mov     DWORD PTR n[rip], edi
35         jmp     inlineme
```

Despite being marked inline, the function is not inlined. Its frame size is set as 520 bytes in the prologue.

If 122 < eax the frame size of `inlineme` is determined by the value in eax.

No stack frame is generated for `main`.

Listing 2: AMD64 assembly for the Example program generated by GCC (LLVM output is practically identical).

For this reason, one of the first steps in symbolizing stack accesses for binary code is to bound the size of each function's call frame. As we saw in the example above even for binaries generated by popular and widely used compiler frameworks with standard compilation options, many functions will not actually have a dedicated call frame, or its size may depend on the context. Functions that are inlined multiple times by the compiler may be inlined into the caller's stack frame with different layouts and sizes. For some functions the size of their stack frames may be a range of possible values depending on the exercised control flow with an upper limit that can be computed at compile time. However, in general we saw that a function's stack frame may not be bounded at all and programs that call `alloca` (or one of its many variants) with a dynamically determined size argument or make use of Variable-Length Arrays (VLAs) can exhibit differently sized stack frames for different inputs. Since this type of runtime-dependent behavior can cause various issues with respect to compatibility, debugging, and memory safety it is often actively discouraged [25].

[2]This is why the Linux kernel requires `-fno-omit-frame-pointer` for producing reliable stack traces.

	`-fstack-usage`	count	`eh_frame`	DWARF info
binary-only	✗	✓	✓	✓
accurate	✓	✗	✗	✗
general	✗	✓	✓	✗
complete	✓	✗	✗	✗

Table 1: Comparison of possible approaches to statically identify stack frame size of a binary function.

Assuming the stack frame size is at least bounded at compile time, there are several ways one could try to statically reconstruct an upper bound for frame sizes of function definitions for a binary program: (1) do the obvious thing and emit stack frame sizes during compilation (`"-fstack-usage"`), (2) count `push` and `pop` instructions, also considering all other modifications of the stack pointer (like `"sub rsp, 504"`) [28], (3) use the entries of the Canonical Frame Address (CFA) column in the function's `.eh_frame` table to try and determine the frame size, (4) collect all `DW_TAG_variable` and `DW_TAG_formal_parameter` entries, as well as registers written to the stack, calculate their respective sizes in bytes, and sum up the total (while also taking their stack offsets into account, which might overlap).

While the first option is probably the safest and most precise, since the compiler originally determines a function's frame layout, this requires source code and target compiler to both be available. It further requires recompilation to be an option for deployment—which may not always be the case even if source code and target compiler are available—and is therefore unfortunately the least generally applicable among all options listed above. In our example both LLVM and GCC correctly report the frame size of `inlineme` as 520 bytes but also mark it as "dynamic".

In contrast, the second option sounds pragmatic, but requires accurate code discovery, disassembly, and modeling of stack operations for each architecture individually—all of which come with their own set of complications. As for more complex binaries and instruction-set architectures a simple linear sweep may not actually be sufficient to discover all stack modifications and symbolically executing functions with global context and intricate control flows would require sophisticated input generation, this approach is constrained to relatively simple cases in practice.

The third option relies on metadata that is present in the vast majority of binaries (including stripped binaries), and hence, may seem like an attractive alternative. However, this requires CFA-entries that are relative to the stack pointer and the compiler may opt to represent CFA entries as base-pointer-relative (or even use general purpose registers), in which case they are useless for determining the stack size of the function statically.

The fourth option requires a debug build of the binary, as well as reliable and complete type information, which is usually not available.

We provide an overview in Table 1: in summary, all of the approaches we discussed are limited with respect to frame size recovery for binary programs and we discuss some of the practical implications of this in Section 5. For our approach, we aimed at

combining the accuracy of a compiler-based solution (which requires source code) with the applicability of binary-only methods to determine frame sizes statically while forgoing the source code requirement. We implemented a simple baseline method for the remaining approaches outlined above as part of our framework to compare against the results using a learned program representation, which we present in the next section.

3 DESIGN AND IMPLEMENTATION

In this section we present the design and implementation of StackBERT. Our main goal is to statically recover stack frame sizes of binary functions. For this, we first train a Machine Learning model based on the popular Transformer architecture using over 600,000 compiled function bodies with masked byte prediction. We then finetune the model to recover an upper bound on each function's frame size.

3.1 Overview

We designed StackBERT to be able to operate in the form of a continuous, supervised learning pipeline that consists of a number of components and present an overview in Figure 1: ① a collection of widely used open-source programs (binutils, coreutils) built with standard compilers (GCC and LLVM) using a number of different optimization levels, ② a label generator, including compiler-based label generation plus tools for parsing ELF binaries using debug information as a baseline implementation for function frame size recovery, ③ a training set of pre-processed and automatically labeled data, ④ a state-of-the-art Transformer model architecture, pre-training task setup, as well as custom finetuning task through binning, ⑤ a dynamic verifier for standard benchmark programs using reference inputs.

3.2 Technical Challenges and Baseline Analysis

As outlined in Section 2, there are several ways in which one might try to obtain a frame size per function statically from a binary in principle. To get a sense of how well such "conventional" approaches perform (and since none of them seemed to be publicly available for general purpose architectures) we prototypically implemented them as a baseline for our approach (cf., "Binary-based Label Generation" in Figure 1). We make several simplifying assumptions such as dealing with benign, compiler-generated executables that are unstripped, formatted as standard ELF files, that may or may not be built using debug information.

First, we utilize the popular `pyelftools` Python library [8] to parse the binaries and read the contents of all executable sections, as well as symbol information (`.symtab`), call frame information (`.eh_frame`),[3] and (if present) debug information (`.debug_info`).

Next, we associate each call frame information with their respective symbol definition. The frame descriptor table consists of entries that are intended to be used for scenarios such as frame unwinding, debugging, and core dumping. An individual entry specifies how the state of the program can be interpreted at any given point during that procedure's execution. For instance, if a register was saved to the stack prior to calling another function the caller's frame table should contain an entry that specifies at which location within the function's frame that value was stored. The syntax for parsing entries is quite complex, due to a finite-state machine encoding for saving space. It is specified as part of the DWARF Debugging Information Format Standard. Although the call frame information in the `.eh_frame` does not represent debugging information, it largely follows the same format.

We parse register rule expressions and generate a preliminary frame layout based on rules that specify a dedicated stack location for any object within the function's frame table. Finally, we collect and propagate any type information that is contained (or can be assumed, e.g., based on the ELF's target architecture) in the binary, to obtain a number of bytes for all stack objects. Although stack slots semantics may depend on the program counter and can, e.g., be reused to hold multiple objects of varying types and sizes at different times, we ignore complex control flows for calculating the maximum frame size. We emit three different estimates for each function's frame size, according to (i) the function's disassembly, (ii) the function's canonical frame address, (iii) the maximally possible sum of all objects with a dedicated stack frame location and well-defined size in bytes. We present the numbers obtained using these three different baseline approaches and a comparison against the estimates obtained using our finetuned Transformer network in Section 4.

3.3 Pretraining: Masked Byte Prediction

The core idea and main component within StackBERT is our pretraining and finetuning of a state-of-the-art Transformer model to learn instruction set semantics in a self-supervised manner, while supporting a variety of different instruction set architectures—like AArch64 and AMD64—as well as potential downstream tasks. In this way, StackBERT is able to readily support prediction tasks largely independently of build toolchain and other metadata typically used in conventional binary analysis approaches. Our generic pretraining setup has two important consequences: first, we are able to leverage the large body of existing software for training without expensive collection of labeled data. Second, the resulting model only requires a minimal amount of information during inference, i.e., binary code of a function.

We pretrain the model using a self-supervised token prediction task, where each token is equivalent to a byte of instruction opcodes as present in the compiled binary function body. Given a sequence of 512 bytes per sample we randomly mask a byte in the sample with a probability of 20%. The model then has to predict the masked bytes in the sample, minimizing over a variation of the cross-entropy loss as objective function, tuned for masked language prediction (see Eq. 1). We refer to Section 5 for additional discussion on our pretraining setup. For pretraining, we use the Adam optimizer with a learning rate of 10^{-4} and polynomial decay scheduling.

$$L = -\sum_i y_i log(\hat{y}) + (1 - y_i) log(1 - \hat{y}_i) \quad (1)$$

[3] We note that although an `.eh_frame` section is not required by ELF, it is not removed by the `strip` tool, and hence, typically present even in stripped binaries. For binaries that do not have an `.eh_frame` section, it can be synthesized automatically [6] from a function's disassembly.

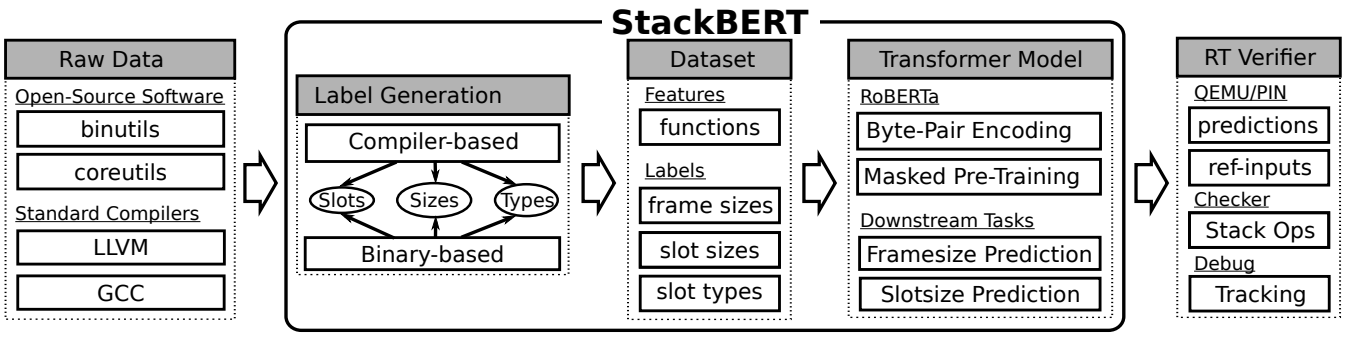

Figure 1: Overview of StackBERT: first, we automatically build and pre-process a large corpus of open-source software using two popular and widely used compiler frameworks. We then automatically extract both features and labels from the compiled artifacts. Next, we pre-train a state-of-the-art Transformer model (RoBERTa), using byte-wise masking of the function disassembly as pretext task. Since the collected label information are not usually present in stripped binaries, we then finetune the pre-trained model using one of our custom downstream tasks, by modeling a key stack symbolization problem as classification with binning. To accurately assess model accuracy we test and validate the model predictions against unseen inputs and also verify its outputs dynamically using standard benchmarks with reference inputs.

3.4 Downstream Task: Frame Size Prediction

For our downstream task, we use the function body as input sequence to our model and the maximum size of its stack frame as output. Bounding the size of a function's stack frame represents an important part of the stack symbolization problem. Potential applications of frame-size recovery include binary instrumentation and rewriting, run-time patching, as well as recompilation after binary lifting to enable use of the native stack. We model frame-size prediction as a classification problem via *binning*: in particular, we define nine different classes that correspond to stack frame sizes in the range of 8 to 2048 (see Section 5 for additional discussion). We use the Adam optimizer with an initial learning rate of 10^{-5} and polynomial decay scheduling to finetune the pretrained model using the *sentence prediction* task, calculating the maximal score of the function body over all classes. We utilize groundtruth labels collected from the two big compiler toolchains GCC and LLVM in recent versions to calculate the loss. We detail our label collection process for this finetuning step in the next section.

4 EVALUATION AND RESULTS

We conduct all of our experiments using Google's Colab cloud compute engines with GPU support—the baseline only uses CPU instances.

4.1 Dataset

To train our model on a representative number of input samples we automatically generate a dataset of compiled binaries and corresponding labels consisting of real-world programs. In particular, we use the popular and widely used `GNU binutils` as well as `GNU coreutils` set of system programs, combining them into our `allutils` corpus of compiled programs (we present an overview in Table 2). We compile both program collections with two mainstream compilers, GCC 11.1.0 and LLVM 13.0.0 and collect groundtruth labels during the compilation process by adding the `-fstack-usage` flag. Compiling with two different compiler frameworks targeting two entirely different ISA's, using a number of varying optimization levels results in a natural diversification of the dataset. However, creating even bigger training sets is entirely possible using our automated data pipeline using these two compilers, possibly including other target architectures. While the outputs of

Task	**Samples**	
Dataset	Binaries	Frame Sizes (Mean/Std)
Training		
allutils-GCC-AMD64-O3	124	36,646 (95.24 / 702.64)
allutils-GCC-AMD64-O2	124	38,115 (85.27 / 684.60)
allutils-GCC-AMD64-O1	124	40,105 (82.17 / 668.48)
allutils-GCC-AMD64-O0	124	52,594 (94.40 / 589.58)
allutils-LLVM-AMD64-O3	124	27,660 (90.16 / 853.00)
allutils-LLVM-AMD64-O2	124	27,783 (89.54 / 851.02)
allutils-LLVM-AMD64-O1	124	27,827 (90.08 / 850.12)
allutils-LLVM-AMD64-O0	124	38,066 (123.81 / 770.62)
allutils-GCC-AArch64-O3	124	41,035 (97.38 / 600.54)
allutils-GCC-AArch64-O2	124	42,865 (87.01 / 582.98)
allutils-GCC-AArch64-O1	124	45,650 (84.50 / 565.93)
allutils-GCC-AArch64-O0	124	60,502 (92.00 / 488.92)
allutils-LLVM-AArch64-O3	124	43,200 (90.49 / 585.35)
allutils-LLVM-AArch64-O2	124	43,512 (89.87 / 583.13)
allutils-LLVM-AArch64-O1	124	43,580 (91.49 / 582.37)
allutils-LLVM-AArch64-O0	124	62,450 (118.97 / 605.01)
Testing		
SPEC2017-GCC-AMD64-O0	23	27,885 (132.95 / 926.94)
SPEC2017-LLVM-AMD64-O0	23	73,627 (92.35 / 645.73)
SPEC2017-GCC-AArch64-O0	23	28,981 (153.31 / 1049.25)
SPEC2017-LLVM-AArch64-O0	23	70,951 (101.42 / 653.41)

Table 2: Overview of our Training and Test Datasets (O1-O3 omitted for brevity on the test set)

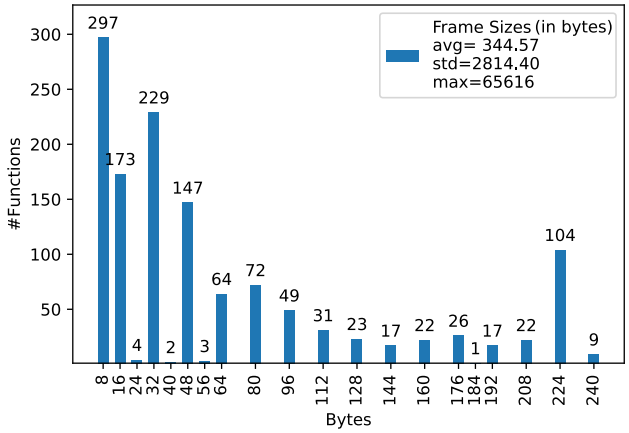

Figure 2: Coreutils Static Frame Sizes (-O3 GCC 11.1.0; x-axis truncated for brevity, since distribution is long-tailed)

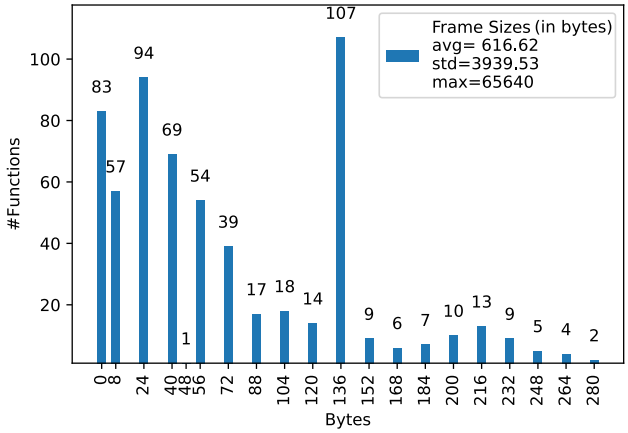

Figure 3: Coreutils Static Frame Sizes (-O3 LLVM 13.0.0; x-axis truncated for brevity, since distribution is long-tailed)

`-fstack-usage` may be unreliable in the presence of link-time optimizations (LTO) [33], SPEC2017 does not compile with LTO by default and does not use it for reference inputs. For this reason, our dataset does not contain any link-time optimized binaries. Each compiler compiles the program for two different architectures AMD64 and AArch64 and across 4 different optimization levels `O0` - `O3`. We design experiments to stress that our approach is compiler and architecture agnostic.

The range of frame sizes in our collected dataset is quite high, with a maximum frame size for optimized binaries of 65640 bytes when compiling Coreutils with LLVM (and 65616 bytes when compiled with GCC respectively), which represents more than 16 full pages of memory. The function allocating this large stack frame is `cksum_pclmul` in `src/cksum_pclmul.c`. [4] Since the distribution of the data is long-tailed (cf., Figures 2 and 3) with very few examples for sizes larger than a couple hundred bytes, we do not use functions with frame sizes over 8192 bytes for training.

We use the SPEC 2017 benchmark suite as a "holdout" dataset for testing our approach. We exclude binaries which involve FORTRAN based code from the evaluation.

4.2 Training Details

As described in Section 3, we use the RoBERTa base model architecture for our experiments as provided in the Fairseq [34] PyTorch library developed by Facebook, which has around 125M trainable parameters. As is common practice for language models with self-attention mechanisms, we first use a pre-training task to automatically learn a useful representation of the raw binary data and subsequently finetune the model on a downstream task.

For the downstream task we use binning with frame sizes spanning multiples of two (starting from 8, 16, 32, 64, etc.) and provide a negative log likelihood loss as defined by the pre-defined sentence prediction task in Fairseq. For each of the models that we train, we use distinct datapoints for the pretraining and finetuning tasks. This ensures that the pretraining task does not get access to any of the labeled data used for the finetuning task.

We pretrain the models for 30 epochs and finetune them for 15 epochs. We observe that the finetuning task converges after 4-5 epochs of training.

4.3 Baseline Results

As explained in Section 3.2 we leverage a combination of `pyelftools` and `dwarf_import` Python libraries to parse the binaries and obtain static estimates of per-function frame sizes using our baseline analysis tool. It is also important to reiterate that our baseline requires unstripped binaries, and will not be able to generate predictions if certain metadata (such as the symbol table and the frame table) is not available. Even with all relevant metadata available the baseline analysis fails to make predictions in a number of cases for our training and test sets due to parsing errors such as unsupported DWARFv5 tags, reaching a total accuracy of 53.86% on SPEC2017 compiled for AMD64. We also evaluate the baseline analysis on SPEC2017 compiled for AArch64, where we observe a substantial drop to less than 20% mean accuracy (cf., Figure 6). The main reason for the substantial drop in performance appears to be the increased usage of DWARF constructs that are not supported in libraries utilized by our baseline implementation, as well as inferior general support for AArch64 binaries. Due to the complexity of the analysis (section parsing, type propagation, frame-table assignment) as well as the overall size of the binaries generating predictions with our baseline for the entire testset takes roughly 45 minutes and consumes up to 12GB of RAM.

4.4 StackBERT Results

Next, we evaluate the finetuned model on SPEC2017 binaries—which were never seen during training. In contrast to the baseline analysis, which relies on metadata embedded in the binary to predict frame sizes, StackBERT is able to generate predictions from the raw binary disassembly of a function body alone. This means it can make predictions even in absence of additional information and

[4] https://github.com/coreutils/coreutils/blob/4edad9e1210dfaa4c8630bad16d0b2e6090de790/src/cksum_pclmul.c#L38

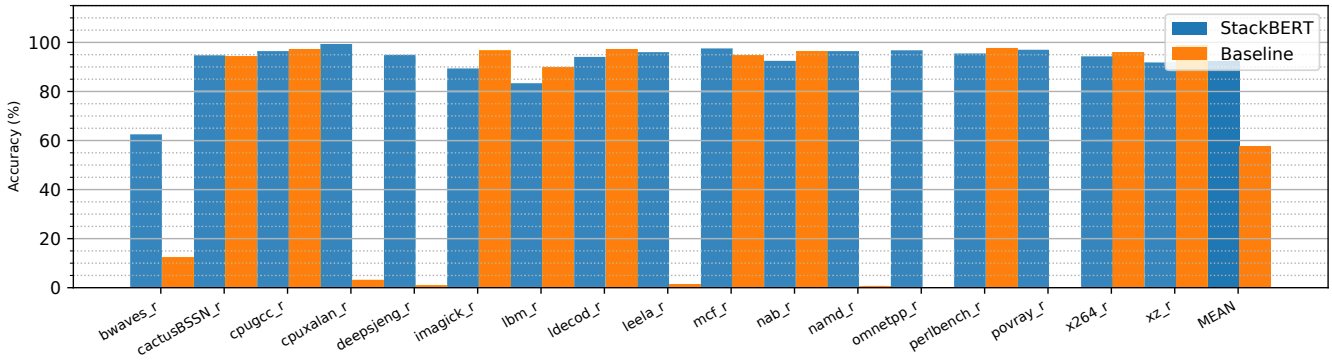

Figure 4: Accuracy of StackBERT vs. Baseline frame size predictions for SPEC2017 (on -O3 AMD64)

only requires raw byte inputs of the binary code, making predictions in a number of cases our baseline cannot cover. We plot the results of its predictions in blue in Figures 4 and 6. On average, StackBERT achieves an accuracy of 93.44% on SPEC2017 compiled for AMD64. Even for highly optimized binaries frame size prediction accuracies never fall below the 50% mark, and drop below 60% for just a single case. On AArch64, StackBERT's mean accuracy remains high at 93.46%.

We would like to highlight that the finetuned model we present here was trained on both architectures, i.e., AMD64 and AArch64. However, during our experiments we also trained a number of dedicated, architecture-specific models to make predictions for each architecture separately. Results for the same are detailed in Table 3.

Interestingly, the jointly trained model achieves an average accuracy that is around 3% higher than its architecture-specific counterpart for AMD64. We conclude that StackBERT manages to learn instruction-set semantics across different architectures automatically through our self-supervised byte prediction pretraining.

While overall inference time depends on the size and complexity of the binary, StackBERT usually manages to predict stack frame sizes for individual functions in a matter of seconds.

4.5 Additional Experiments

We conducted additional experiments to evaluate the fidelity of the learned representation by deliberately modifying stack sizes of binary function bodies. We chose the `mcf` binary from SPEC 2017 compiled with `O0` optimization using GCC 11.1.0 for AMD64 for this experiment. From this binary, we randomly picked function

Table 3: Mean accuracies across the SPEC2017 benchmark suite for the different models across optimization levels

Model Type Evaluation Dataset	AMD64 AMD64	AArch64 AArch64	Unified AMD64	Unified AArch64
O0	90.35%	96.58%	93.56%	96.11%
O1	92.72%	95.73%	95.32%	92.85%
O2	90.87%	94.85%	92.38%	92.50%
O3	91.34%	93.75%	92.49%	92.38%
mean	91.32%	95.23%	93.44%	93.46%

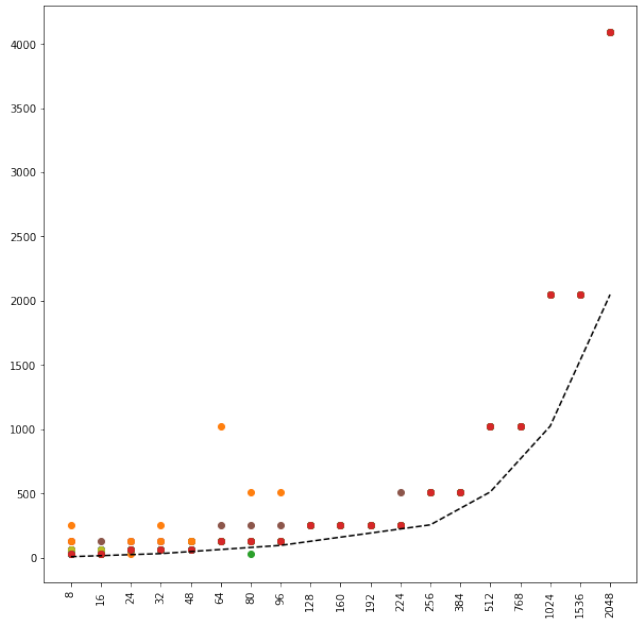

Figure 5: Predictions made by StackBERT under targeted modification of frame sizes continue to remain correct.

bodies containing direct manipulation of the stack pointer (e.g., `sub rsp, 0x40`). We modified the operand of these instructions by subsequently drawing integer values from a fixed range of numbers between 8 and 2048. We then predict the function's stack frame size after each individual modification. The result is depicted in Figure 5. The dashed black line shows the range of values that was drawn from the list. The dots show the predictions by StackBERT; as depicted only 0.22% of the predictions lay below the dashed line, meaning that a vast majority of the predicted values remain correct. We conclude that the trained model accurately learns which instructions will affect frame size through our pretraining and finetuning tasks. We also infer that the trained model learns quantitative relationships of tokens within the instruction and their relevance for the frame size of the containing function.

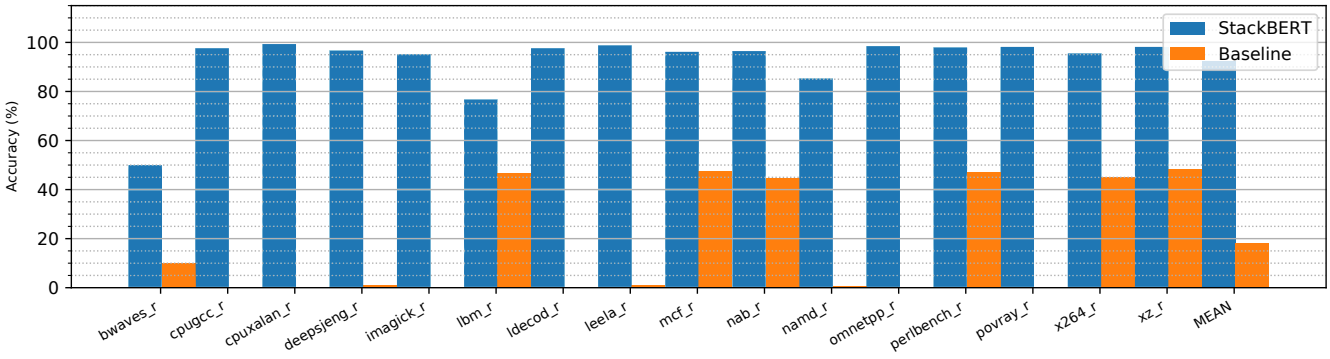

Figure 6: Accuracy of StackBERT vs. Baseline frame size predictions for SPEC2017 (on -O3 AArch64)

Last but not least we verified all of the correctly labeled predictions made by StackBERT for the SPEC2017 binaries by executing the respective binary and recording the stack frame size for each executing function using the PIN tool. We run the binaries with the test input provided in the benchmark suite to validate frame sizes of functions covered as part of the trace. While this severely impacted the execution time of the program, we were able to verify correct predictions of each of the function's frame size by checking that none of the static estimates exceeded the recorded runtime values.

5 DISCUSSION AND FUTURE WORK

In this Section we briefly discuss assumptions, pretraining, downstream task and binning, as well as possible applications.

As mentioned in the beginning, static binary analyses typically range from complex to generally unsolvable [20, 43]. For this reason, we constrain the stack symbolization tasks mentioned in this paper by making several basic assumptions: we assume standard ELF files that can be loaded by general purpose operating systems (i.e., the ELF magic is set and the file format conforms to the existing standards). We further assume that the binary files contain several standard information, such as a symbol table and correct function boundary information. While this may not always be the case in practice (for instance, because the binary was stripped) function boundary identification and symbol recovery are complimentary to StackBERT and several existing approaches demonstrate that such information can be recovered from stripped binaries in practice [1, 5, 37, 41]. We refer to Section 6.2.1 for an overview of existing methods. For our baseline analysis we further assume that the machine architecture matches the target architecture specified in the ELF file and that section headers and parameter settings are provided and accurate. While StackBERT does not rely on any metadata in the binary, our baseline analysis checks for a number of binary features in ELF files containing debug information, such as inlined calls, nested inlining, type information, variable and parameter declarations, and the baseline analysis assumes all of these to be correct.

For pretraining, it is noteworthy to mention that we also experimented with an instruction-based pretraining method, where we applied a similar objective as mentioned in Section 3.3 to disassembled byte sequences by masking all bytes belonging to an individual instruction at the same time. However, this instruction-wise pretraining method actually performed worse in our initial tests compared to the byte-wise masked pretraining in the subsequent downstream task. For this reason, we used the masked language prediction task as desdribed above. Similar to related work [37] we hypothesize that pretraining lets the model learn an instruction-set architecture's (ISA) semantics prior to identifying the task-relevant bits, such as the instructions and operands that relate to the stack pointer. We note that truncating inputs to 512 bytes is a consequence of the underlying model architecture. In our case, this is not problematic as compilers generate instructions that define a function's frame size in the function prologue, i.e., at the beginning of the function body typically within the first few dozen bytes. However, should a function prologue exceed this limit in the future we could adopt model variations [7, 23, 46] that forgo input truncation.

Our downstream task uses binning to model stack frame size recovery as a classification problem. Our rationale for binning stack frame size predictions is two-fold: first, due to data alignment and performance optimizations compilers already tend to favor certain stack frame sizes over others. Second, language Transformer models with self-attention (like RoBERTa) are known to exhibit limitations with regards to counting and regression [9, 18]. However, there is no fundamental reason for binning frame sizes and we believe that with additional implementation and experimentation a regression model for frame size prediction (possibly using a modified architecture) should be feasible in principle.

Ideally, we would like to extend our existing downstream task to predict additional information about a function's stack frame, such as the number of individual objects and their respective sizes. While we anticipate StackBERT's design to be able to handle these additional tasks without requiring any modifications, the main challenge for this extension lies in the ability to gather highly accurate ground truth data. Our baseline analysis contains preditions for detailed stack layouts, however, upon manual inspection of the results their quality appears to be in similar (if not worse) condition than the baseline frame size predictions. Unfortunately, we also found that current compilers provide an absolute minimal amount of information about a function's stack layout and gathering additional

information would likely require a modification of the compiler toolchains. While certainly possible in principle we have to leave this task as an interesting avenue for future work.

Finally, we mentioned in the beginning that binary patching, rewriting, and lifting are important motivations for our work. For instance, current state-of-the-art lifting approaches rely on emulated stack environments for recovered binaries [2, 15]. This means that stack operations in recovered binaries will typically be modeled as indexed accesses into a dynamically allocated, global array variable, severely impacting runtime performance from 1.5x to up to 3x compared against original binary executions that utilize the native stack. We envision our results to be useful for these applications, since knowing a function's frame size while lifting it would enable breaking the global stack array into function-local arrays. Knowing an upper bound on the function's frame size should suffice to enable this use case: binary lifting frameworks can use this information to lower emulated, function-local stack frame arrays during recompilation to utilize the native stack, potentially yielding huge performance gains. According to our evaluation results in Section 4 more than 90% of binary functions in SPEC2017 could potentially have been rewritten to make use of the native stack in the recovered binary, rather than emulating stack accesses.

6 RELATED WORK

Automated software analysis can be surprisingly difficult. While fundamental concepts are easily explained answering questions about the runtime behavior of a given piece of code with certainty may be impossible [20]. Working with binary code aggrevates those challenges, since a lot of semantic information about the program that may be specified and present at the source level does not actually represent meaningful input for the hardware and is thus discarded during the compilation process. Nonetheless, automated static analysis of binary programs has been studied for nearly three decades and a large body of literature on the topic exists. In the following, we provide a brief overview and comparison against relevant approaches.

6.1 Conventional Machine Code Analysis

6.1.1 Binary Instrumentation and Rewriting. The idea of analyzing and modifying binary programs is quite old. While early approaches aimed at identifying control flow and intra-procedural data dependencies, several works soon started investigating practical methods of instrumenting the binary in question. One of the early approaches is EEL [26] which aimed at simplifying binary editing by providing architecture-independent abstractions. Several other approaches such as DynamoRIO [11, 12], PIN [30, 39] and Valgrind [32] proposed to instrument binaries during execution, which is comparatively safer and more reliable since it avoids most of the problems of static code discovery and analysis.

While binary rewriting approaches were initially rather limited in their capabilities, progress in binary analysis techniques allowed more recent binary rewriting frameworks to provide a rich set of functionalities with support for many different architectures and even including use cases like automated binary hardening transformations [35, 40, 47, 48]. Several tools aim at analyzing stack accesses using architecture-specific knowledge:

PLTO [28] by Linn et al. analyze stack sizes for binary functions in standard benchmarks using relocation information. However, they do not analyze the accuracy of their recovered estimates.

The Stacktool [40] was developed to prevent stack overflows and bound overall stack size on AVR-based systems, but focuses on embedded systems and does not support complex instruction set architectures.

Unfortunately, general static analysis approaches for binaries encountered significant challenges with respect to code discovery, memory accesses, and recompilation, leading to the idea of binary lifting [2, 4, 15].

6.1.2 Binary Lifters. The idea behind binary lifting is to analyze executable binaries with respect to control flow as well as memory accesses and generate an in-memory intermediate representation (IR) suitable for further processing—similar to how compilers use IRs to reason about programs while translating source to machine code. Binary lifting approaches such as CodeSurfer [3], BAP [13], McSema [15], and BinRec [2] currently represent the state-of-the-art in binary program analysis and instrumentation. However, due to the strict functionality requirements of these binary lifters (e.g., for the purpose of recompilation) they rely on abstractions and helper constructs that the originally compiled program did not have: for example, they usually provide an emulated stack environment using a large, dynamically allocated array, as well as various fallback mechanisms to the original binary code. Our goal is to improve static analysis of binary programs to aid in alleviating some of those restrictions in the future.

6.2 Machine Learning for Machine Code

Machine learning (and in particular "Deep Learning" [27]) demonstrated significant progress over the past 10 years in a number of challenging and complex domains such as natural-language processing, computer vision, and robotics that have traditionally proven difficult for conventional software. It is thus perhaps unsurprising to see increasing adoption of ML-based approaches in other domains and a number of recent approaches propose to apply ML to static program analysis tasks. Since this area is rapidly growing we focus on a direct comparison with approaches that explicitly target *binary analysis* in this section.

6.2.1 Binary Function Identification. Several works tackled binary function identification using ML:

Byteweight [5] was one of the earliest examples that aimed at a data-driven approach to binary analysis. They propose a learned prefix tree representation for disassembly with normalized immediate values to classify function start addresses, tackling the function identification problem and beating IDA Pro (the state-of-the-art at the time) by an order of magnitude, establishing a new baseline.

Shin et al. [41] were the first to demonstate neural networks for function identification, showing overall improvements in training and inference time over Byteweight (which was not based on neural networks) while keeping similar performance using a recurrent architecture with byte-wise one-hot encoding. Function identification approaches are complimentary to StackBERT, since we assume function boundaries to be known (e.g., through `.symtab`).

Gemini [45] proposes to train a control-flow-based embedding for the purpose of binary function similarity detection, demonstrating order of magnitudes speedup in training and inference, as well as improved accuracy. Function similarity detection is orthogonal to our approach, but generally useful for malware classification and debloating.

FUNCRE [1] tackles function inlinining detection by following up on an earlier approach [37]. Detecting if a function was inlined at a particular code location represents an important sub-task in analyzing highly optimized binaries. They are able to improve the F-score of state-of-the-art approaches in function inlining detection by about 3%.

Since we assume function symbols to be known for our frame size and layout prediction tasks, function identification approaches are complimentary to StackBERT.

6.2.2 Decompilation. An increasing number of ML-based binary analysis approaches aims to recover source-level information (commonly called "decompilation"), such as variable and function names, function signatures and line numbers, as well as high-level code constructs like loops, conditions, and switch statements:

Debin [19] uses a lifted binary intermediate representation to predict source-level debug information. They propose a conditional-random-field-based graphical model using factor graph representations and bayesian inference for structured prediction of likely source-level variable types and names given a particular binary program.

Coda [16] presents a neural network architecture to decode a binary into an Abstract-Syntax Tree (AST) of a high-level source language, that is then iteratively refined using the target binary in a second step, achieving 82% program recovery accuracy on short, custom benchmarks built without optimization (-O0).

N-Bref [17] proposes a dedicated structural transformer architecture using an assembly encoder, an AST encoder, and decoder. They demonstrate improvements of 6.1% and 8.8% accuracy in datatype recovery and source code generation respectively using short program snippets.

Punstrip [36] combines probabilistic fingerprinting with a graphical model to learn relationship between function names and binary features. They are able to predict semantically similar function names based on code structure for standard library functions.

Perhaps closest to our approach is XDA [37], which proposes a transfer-learning-based disassembly framework that uses masked language modeling as a self-supervised pre-text task to decompilation. The authors evaluate their approach on function boundary identification and code discovery as downstream tasks, using standard benchmarks on x86 and AMD64, achieving 99.0% and 99.7% F1 scores respectively.

Decompilation is orthogonal in principle to StackBERT, since our main goal is to recover low-level stack memory layout of a given binary function—which none of the existing approaches target, support, or evaluate.

6.2.3 Binary Type Inference. Another line of work is learned type inference for binaries. For example, Eklavya [14] proposes use of neural networks for function signature recovery by solving two tasks: function argument count and argument type recovery. They achieve an accuracy of 84% and 81% for both tasks respectively using a recurrent architecture with an instruction-wise, skip-gram-based word embedding model that is trained separately. Function signature recovery partially overlaps with function frame recovery, as parameters may be passed via the stack. However, in practice many optimized functions do not receive parameters via stack accesses but through registers instead to increase runtime performance. Inferring types for local variables—which occupy the majority of function frames in our dataset—is not supported by Eklavya.

6.2.4 Learned Models for Binary Execution. Finally, Ithemal [31] proposes to model execution timing aspects of complex instruction set architectures statically using an LSTM-based neural network architecture. They demonstrate that cycle-accurate throughput predictions can be learned efficiently for modern microarchitectures, significantly improving over the prior state-of-the-art (which does not use machine learning). Modeling timing aspects of instruction sequences represents an interesting but orthogonal task in static binary analysis.

7 CONCLUSION

We present StackBERT, a novel binary analysis approach tailored towards key stack symbolization tasks like frame size recovery. We demonstrate byte-wise pretraining and custom finetuning tasks that result in high-accuracy predictions on standard benchmarks, that were not seen during training. We validate model prediction correctness using both compiler-generated labels and runtime information. In comparison with conventional baseline binary analysis approaches StackBERT performs favorably and predictions remain stable across high optimization levels, different compiler toolchains, and different architectures. We conduct experiments that indicate StackBERT is able to learn instruction-set semantics completely automatically through self-supervised pretraining and anticipate our findings to be useful for a number of additional downstream tasks in the future.

ACKNOWLEDGMENTS

We thank all reviewers for their comments. Thanks to Fabian Parzefall, Min-Yih Hsu, Dokyung Song, Tristan Ravitch, Ledah Casburn, and Daniel Matichuk for helpful discussions and general feedback. This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contract N66001-20-C-4027, and by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, or any other agency of the U.S. Government.

REFERENCES

[1] Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. 2021. Finding Inlined Functions in Optimized Binaries. https://arxiv.org/pdf/2103.05221.pdf. (2021).

[2] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[3] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.

[5] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.

[6] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and fast DWARF-based stack unwinding. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, Vol. 3. ACM New York, NY, USA, 1–24.

[7] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[8] Eli Bendersky. 2011. pyelftools: Parsing ELF and DWARF in Python. https://github.com/eliben/pyelftools. (2011).

[9] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264* (2020).

[10] Eric Botcazou, Cyrille Comar, and Olivier Hainque. 2005. Compile-time stack requirements analysis with GCC. In *Proceedings of the 2005 GCC Developer's Summit*. Citeseer, 93.

[11] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, transparent, and comprehensive runtime code manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering âĂę.

[12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 133–144.

[13] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[14] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. 99–116.

[15] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*.

[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, Vol. 32. 3708–3719.

[17] Cheng Fu, Kunlin Yang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. 2020. N-Bref: A High-fidelity Decompiler Exploiting Programming Structures. (2020).

[18] Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics* 8 (2020), 156–171.

[19] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1667–1680.

[20] R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *Comput. J.* 23, 3 (1980), 223–229.

[21] The kernel development community. 2021. Kernel livepatching consistency model—stack checking. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/livepatch/livepatch.rst#n97. (2021).

[22] The kernel development community. 2021. ORC unwinder. https://www.kernel.org/doc/html/latest/x86/orc-unwinder.html. (2021).

[23] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451* (2020).

[24] Mitja Kolsek and the 0patch Team. 2017. Did Microsoft Just Manually Patch Their Equation Editor Executable? Why Yes, Yes They Did. (CVE-2017-11882). https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html. (2017).

[25] Michael Larabel. 2012. The Linux Kernel Is Now VLA-Free: A Win For Security, Less Overhead and Better For Clang. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kills-The-VLA. (2012).

[26] James R Larus and Eric Schnarr. 1995. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. 291–300.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[28] Cullen Linn, Saumya Debray, Gregory Andrews, and Benjamin Schwarz. 2004. Stack analysis of x86 executables. *Manuscript. April* (2004).

[29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[31] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*. PMLR, 4505–4515.

[32] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *SIGPLAN*.

[33] Chromium OS. 2017. Stack Size Analyzer. https://www.chromium.org/chromium-os/ec-development/stack-size-analyzer. (2017).

[34] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019).

[35] Vasilis Pappas. 2012. kBouncer: Efficient and transparent ROP mitigation. *Apr* 1 (2012), 1–2.

[36] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*. 373–385.

[37] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. In *Symposium on Network and Distributed System Security (NDSS)*.

[38] Josh Poimboeuf. 2016. objtool: add tool to perform compile-time stack metadata validation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=442f04c34a1a467759d024a1d2c1df0f744dcb06. (2016).

[39] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. 2004. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. 22–es.

[40] John Regehr, Alastair Reid, and Kirk Webb. 2005. Eliminating stack overflow by abstract interpretation. In *ACM Transactions on Embedded Computing Systems (TECS)*. ACM New York, NY, USA, 751–778.

[41] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 611–626. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[42] Linux Torvalds. 2012. Re: [RFC 0/5] kernel: backtrace unwind support. https://lkml.org/lkml/2012/2/10/356. (2012).

[43] A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230 arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[45] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[46] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big Bird: Transformers for Longer Sequences.. In *NeurIPS*.

[47] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[48] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*. 337–352.