# Learning to Find Usages of Library Functions in Optimized Binaries

Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant

**Abstract**—Much software, whether beneficent or malevolent, is distributed only as binaries, sans source code. Absent source code, understanding binaries' behavior can be quite challenging, especially when compiled under higher levels of compiler optimization. These optimizations can transform comprehensible, "natural" source constructions into something entirely unrecognizable. Reverse engineering binaries, especially those suspected of being malevolent or guilty of intellectual property theft, are important and time-consuming tasks. There is a great deal of interest in tools to "decompile" binaries back into more natural source code to aid reverse engineering. Decompilation involves several desirable steps, including recreating source-language constructions, variable names, and perhaps even comments. One central step in creating binaries is optimizing function calls, using steps such as inlining. Recovering these (possibly inlined) function calls from optimized binaries is an essential task that most state-of-the-art decompiler tools try to do but do not perform very well. In this paper, we evaluate a supervised learning approach to the problem of recovering optimized function calls. We leverage open-source software and develop an automated labeling scheme to generate a reasonably large dataset of binaries labeled with actual function usages. We augment this large but limited labeled dataset with a pre-training step, which learns the decompiled code statistics from a much larger unlabeled dataset. Thus augmented, our learned labeling model can be combined with an existing decompilation tool, Ghidra, to achieve substantially improved performance in function call recovery, especially at higher levels of optimization.

**Index Terms**—Reverse engineering, Software modeling, Deep learning



## 1 Introduction

In their seminal work, Chikofsky and Cross [1], define *Software Reverse Engineering* as "the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction". Understanding the behavior of software binaries generated by potentially untrusted parties have many motivations such binaries may incorporate *e.g.*, stolen intellectual property (such as patented or protected algorithms or data), unauthorized access to system resources, or malicious behavior of various sorts. The ability to reverse engineer binaries would make it more difficult to conceal potential bad behavior, and thus act as a deterrent, and this would enhance public confidence in, and overall free exchange of, software technology.

Reverse engineering of an *optimized* binary is quite challenging, since compilers substantially transform source code structures to create the binary. This is done primarily to improve run-time performance; however, some compilers support deliberate obfuscation of source code details, to protect IP, or for security reasons. The resulting binary could be stripped of all information such as variable names, code comments, and user-defined types. Binaries compiled using `gcc` can be optimized in the interest of run-time performance benefits, (even if compilation *per se* takes longer). Optimizations include function inlining, array vectorization, and loop unrolling. This dramatically alters the code at the assembly level, making it substantially more challenging to decompile the binary successfully.

Two tools are industry standard for reverse engineering: Hexrays IDA Pro [2] and Ghidra [3]. These tools incorporate two distinct functionalities: a *disassembler* (convert binary to assembly code), and a *decompiler* (convert assembly code to a higher-order representation, similar to C code), and a variety of data flow analysis tools. Both tools can handle binaries that have been compiled on a variety of architectures (such as x86/64 or ARM64).

In the field of security, quite a bit of work has focused on understanding the behavior of malicious applications by examining their library API calls [4], [5], [6], [7], [8], [9], [10], [11], [12]. The intuition behind this is that calls made to library APIs (such as Windows DLLs) can capture the important underlying semantics of the malware's attacking behaviour [13]. However, uncovering API calls is particularly hard as the compiler might have mangled the inlined body of the *called* function together with the code at the *calling* site in complex ways. Both Ghidra and Hexrays have specially engineered functionality for recovering calls to library functions . These functions are considered very important by the developers of these tools and are explicitly documented and advertised.

Both solutions use a database of assembly-level signatures for each potentially inlined library call. Assembler code recovered from the binary is matched against the database to identify library function bodies within the binary. The approaches, however, do not work as well with higher optimization levels due to extensive code restructuring [14], [15] and context-dependent inlining of the called method's body. Despite both tools making an attempt to find library function invocations in a static way (see, *e.g.*, [16], item 7), the problem is non-trivial and more often than not

• *All the authors are with the Department of Computer Science, University of California, Davis, CA, 95616.*
*E-mail: {tfahmed, ptdevanbu, asawant}@ucdavis.edu*

*Manuscript in submission*

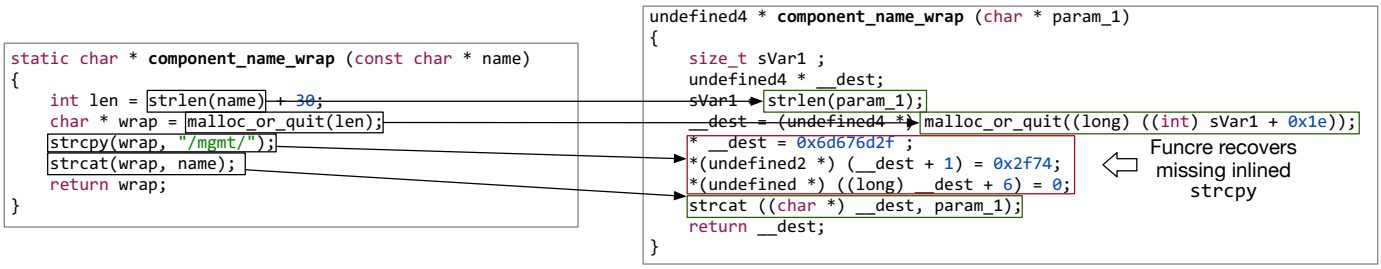


Fig. 1: Finding inline functions in real-world[1]decompiled version of original C source code

these functions are not recovered. This problem is complicated by varying compilers, compiler versions, optimization levels, and library versions.

This signature or pattern-oriented approach adopted by current tools relies on a pattern database; this database must be *manually maintained* to include a pattern (or patterns) for each possible inlined function. Therefore, we adopt a more data-driven approach and propose to *learn* to identify those functions that are most often used in the data. To that end, we develop our tool, FUNCRE, which finds inlined library function invocations even in binaries compiled with higher optimization levels. With sufficient data, a suitably designed training approach, and a powerful enough model, this approach offers enhanced ability to recover inlined functions. As an illustration, in Figure 1, we present a sample of original source code (from GitHub) and its Ghidra output for compilation with Os. From the decompiled version, it's evident that recovering `strlen`, `malloc_or_quit` and, `strcat` is possible for Ghidra, but the `strcpy` gets inlined with copying of pointers to stack addresses after optimization. The pattern-database approach used by Ghidra works well for the three of the four functions; but Ghidra fails to recover the latter, (*strcpy*), because the more complex pattern required is not available in its database. The precision of Ghidra is 1.0 for this example, but the recall is lower (0.75). Our tool FUNCRE builds on Ghidra, and will only try to recover additional inlined functions beyond what Ghidra does, using *automatically learned patterns* latent within its layered deep-learning architecture. In this specific case, FUNCRE is able to recover the `strcpy`, thereby improving Ghidra's recall to 1.0. FUNCRE, can do this because it can *learn* to find the patterns (within decompiled code) that reflect the occurrence of the various possible inlining patterns of functions within the decompiled source of optimized binaries.

We note that our tool, FUNCRE, *starts with*, and *builds upon* the output of Ghidra decompiler. It works on top of Ghidra-decompiled source code output; it tries to recover *only those* functions missed by Ghidra. Our training dataset consists of *decompiled* binaries (built at scale by compiling open-source code) with labels indicating the position of inlined function calls contained within, specifically those inlined functions which the decompiler *failed* to recover. We created a build-pipeline that leverages Docker and Travis CI; we successfully built binaries for almost 1,200 projects. We then decompile these binaries using Ghidra to obtain a learned vector representation wherefrom we recover inlined functions. To obtain labeled data (*i.e.*, knowing which function has been inlined where), we develop a custom labeling approach that allows us to places labels (with the name of the function) in the binary, at locations where a function has been inlined. Our labels *resist* being removed by optimizing compilers; this also minimizes interference with the optimizing compilers code generation. This labeling approach allows us to successfully annotate 9,643 files and 459 open-source projects, creating a large training dataset.

Our approach uses a pre-training plus fine-tuning scheme [17] currently popular in natural language processing. We pre-train using a masked language model (MLM) on decompiled binaries. This pre-training step helps our model learn the statistics of a large corpus of decompiled source code. These statistics are represented in a low-dimensional positional embedding. We then fine-tune on the labeled dataset of inlined functions. FUNCRE achieves a precision of 0.64 and a recall of 0.46. When combined with the results of Ghidra (since it uses the output of Ghidra, which might contain some functions recovered), our approach provides the highest f-score across all optimization levels while also recovering a larger number of unique functions. Recovering inlined functions (where assembly-level function prologues & epilogues are absent, the inlined body is changed and shuffled by optimization) is hard even for humans and existing tools; however, a high-capacity neural model (RoBERTa + Transformer) that uses pre-training and fine-tuning can still learn statistical features and associations between inlined functions and the mangled residual clues that signal their presence. This enables the recovery of 19% more unique functions over Ghidra and 10% over Hexrays. We make the following contributions:

1) We have generated a training set of 670K fragments for fine-tuning for inline function recovery from 8.6K binaries, each having at least one inline function. We also have a validation set of 25K fragments (from 270 binaries) and a separate test set for each optimization level.
2) We pre-train a RoBERTa [17]architecture with an enormous amount of unlabeled decompiled binary data. We fine-tune it for the downstream task inline function recovery and achieve state-of-the-art performance. We also show that Transformers with pre-trained embedding works better than standard Transformers proposed by Vaswani *et al.* [18].
3) We improve the F-score of library function recovery by 3%-12% at different optimization levels without introducing a significant amount of false positives.

## 2 Background

### 2.1 Importance of function calls in binary comprehension

Eisenbarth *et al.* [19] and Schultz *et al.* [4] argue that identifying library function calls made within a binary are key to comprehending its behavior. Substantial prior research in the field of binary analysis has focused on this problem.

Much of the effort to understand binaries is to identify malware. Schultz *et al.* [4] use the calls made to Windows system APIs to understand if a binary has malicious intentions. Ye *et al.* [6] found that reverse engineers can identify malware in a binary based on the co-occurrence of six calls made to a specific kernel API.

A barrier to static analysis techniques is that sometimes binaries can be optimized and/or obfuscated. To overcome this, researchers have used dynamic analysis to understand the APIs being accessed by a binary. Hunt and Brubacher [20] and Willems *et al.* [21] attempt to detect calls made to Windows system APIs by instrumenting the system libraries. Bayer *et al.* [22] and Song *et al.* [23] emulate the Windows runtime and recover the Windows system API calls. In comparison to static analysis, dynamic analysis is limited by test set coverage, as well as by dynamic cloaking (malware could disguise its behavior when it knows it being surveilled, *e.g.*, in a VM).

Given how important library/API calls are to reverse engineers' understanding of the semantics of a binary, it is pivotal that these calls are recovered by disassembler and decompiler. However, optimizing compilers can inline many library calls, thereby making them hard to recover, even by state-of-the-art tooling; improving library function recovery is an important problem.

### 2.2 Disassembler vs decompiler

Binaries are created in two stages: (1) source code is preprocessed and compiled into machine code and (2) the machine code is linked with all supporting code such as libraries and system calls to create the executable binary. Similarly, the process of reverse engineering of a binary comprises of two stages: (1) the binary is "disassembled" into assembler code and (2) the assembler code is converted into a higher-order representation which is close to the original source code.

*Disassemblers* such as Ghidra, Binary Ninja, IDA Pro, and gdb perform the first stage of reverse engineering. Since the machine instructions in a binary generally have a one-to-one mapping with the assembly instructions for most platforms, disassembly *per se* is relatively straightforward.

Next, *decompilers* such as Ghidra, Hex-rays, and Snowman transform the machine code produced by the disassembler into a more legible representation referred to as pseudo-C code. Decompilers produce code that is ostensibly more compact and readable than that produced by a disassembler. They also often recover the original function boundaries (*i.e.*, where function *bodies* start and end), control flow structure, and primitive type information. This makes decompilers a better tool for a reverse engineer as the effort and knowledge required to understand pseudo-C code is less than that of reading assembler code.

### 2.3 Library function recovery

Given its importance, we focus on the recovery of function calls from binaries. The two main reverse engineering tools - Hex-rays IDA Pro (commercially available with a license cost of approximately $10,000[2]) and Ghidra (open source and maintained by the National Security Agency) - have dedicated solutions targeted just for recovering function calls. The developers of Hexrays acknowledge the importance of function recovery by stating that: "Sometimes, the knowledge of the class of a library function can considerably ease the analysis of a program. This knowledge might be extremely helpful in discarding useless information." [14]

Both tools can identify function calls. They maintain a database of function signatures at the byte level (assembler code). They recover the function by checking each sequence or block of operations in the disassembled code against the signatures in the database.

We observe that majority of the previous research work in this field is based on call graph matching which has been designed to be robust to statement interleaving due to compiled optimization. These approaches are static in nature and try to go beyond the offerings of Ghidra and Hex-rays.

Qiu *et al.* [15], [24] implement a static approach to recover inlined library functions by extracting execution dependence graphs for a library function and then matching this in decompiled code to recover. This work reports deal with inlined functions in optimized binaries, however, the evaluation lacks a performance breakdown by optimization level. Furthermore, only precision numbers are reported on a small subset of inlined string library functions, and the overall performance is not compared to Ghidra or Hex-Rays.

BinShape by Shirani *et al.* [25] also uses graph features for function identification. However, they do not assess the efficacy of their approach against inlined functions. "impact of inlined functions" were not scrutinized. We are the first to attempt this task with a neural model and compare this to the SOTA tools such as Ghidra and HexRays.

There have been a couple of neural approaches to recovering function names but not inlined functions. He *et al.* [26] present a tool called Debin that is based on a combination of a decision-tree-based classification algorithm and a probabilistic graph model that attempts to recover function invocations and other symbols in obfuscated code. David *et al.* [27] encode the control flow graphs of invocation sites and try to recover the correct invocation name using LSTM's and Transformers. Neither approach explicitly deals with inlined library functions nor present any results broken down by optimization level.

The function recovery in these tools has a major flaw: they aren't good at recovering functions from binaries that have been compiled with optimizations. In C and C++, there are six optimization levels (in increasing order of complexity: O0, O1, Os, O2, O3, and, Of). Code compiled with O0

1. Original source from GitHub: https://github.com/rlite/rlite/blob/aaec531682756f17c36249e71013a5d3b4f374f9/user/tools/rlite-ctl.c#L1064

2. Estimate based on the cost of base IDA Pro disassembler license and the cost adding three platform-specific Hexrays decompilers

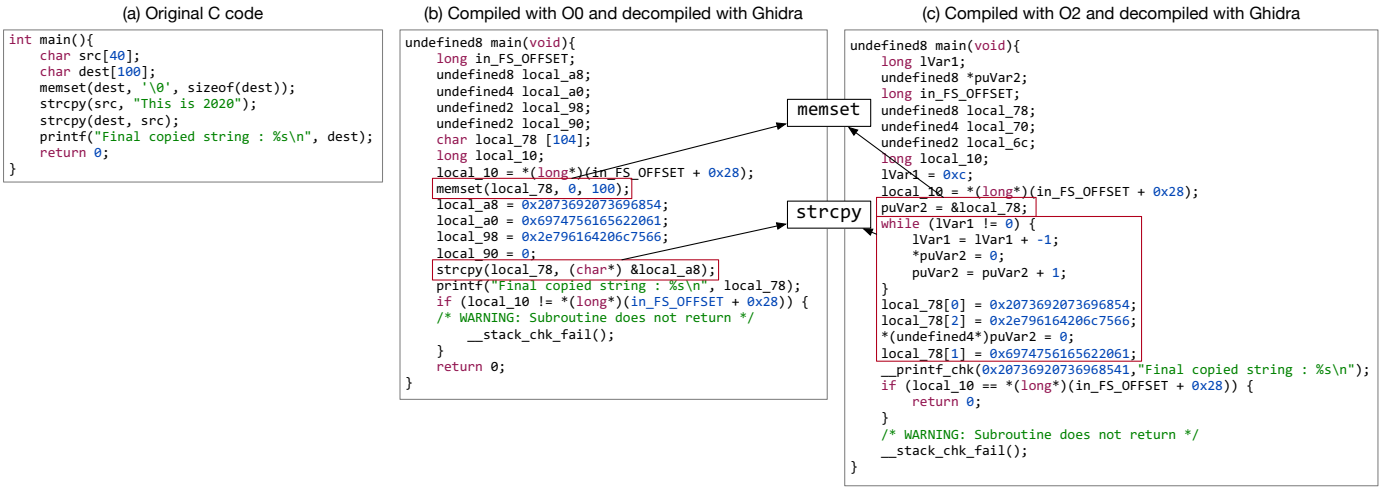


Fig. 2: Comparison between original source code (a), Ghidra output for compilation with O0 (b) and Ghidra output for compilation with O2 (c)

is the most basic: compilation is fast, and no optimization is done. Starting from O1, more and more optimizations are introduced, and the code structure changes substantially. At Of, the most optimized level, the compiler does not even guarantee correctness. Hex-rays IDA Pro and Ghidra work better with code that has been compiled using the O0 or O1 optimization levels, since the code structure is largely preserved.

In the toy example seen in Figure 2(a), we see that the source code of a file is written in C. The function depicted invokes `memset` and then `strcpy` (twice). When we compile this file with no optimizations (the O0 flag) and then decompile it using Ghidra (output seen in Figure 2(b)), we see that the decompiler can recover called functions and can create a generally good representation of the original C code. Note, however, that it "fuses" the chained string copy invocations. When we compile with a higher optimization level such as O2 and then decompile the file, we see the result in Figure 2(c). The performance of the decompiler degrades, as the binary gets more optimized: some library function uses are no longer recovered. In the figure, we highlight the parts of the function relating to the library function calls, whose implementation has been inlined. In this example, we do see that three other function calls are recovered, however, this could be due to the fact that they were never inlined or that Ghidra does a good job of recovering them even if they were inlined.

We want to clarify that like [28], [29], [30], [31] we do not target the function boundary identification task. Ghidra & Hexrays already do this at 90%+ accuracy. They do much worse at the recovery of inlined library functions; This task is a challenge for the heuristic method used by Ghidra & Hexrays, especially at higher optimization levels, as acknowledged by the developers [2]; by leveraging powerful neural models (explained in Section 4), FUNCRE can improve these tools.

A decompiler can recover most of the semantics of the original code, however, it has a hard time recovering variable names, struct types, exact data flow, code comments, and inlined library functions. Most of this information is lost in the compilation - decompilation loop, *e.g.*, in Figure 2 the decompiler adds a lot of new local variables each with an ambiguous name.

State-of-the-art approaches to improving decompilation employ machine learning (ML) techniques. Katz *et al.* [32] propose to decompile disassembled code by using a Neural Machine Translation (NMT) model. Their approach currently works at the statement level and seeks to recover natural C code for each block of disassembled code. Lacomis *et al.* [33] use an encoder-decoder model to recover variable names. Their approach only targets code compiled with O0 and not on higher optimizations.

We see that ML approaches to improving decompilation are limited. We hypothesize that an ML-based approach will work well for the task of library function recovery because ML can detect patterns in highly unstructured data.

## 3 Approach/Methodology

### 3.1 Research questions

Our approach to improving library function recovery builds on top of the pseudo-C code produced by the Ghidra decompiler. Using large volumes of source-available projects and some careful, automated instrumentation, we develop a *supervised learning* approach to find library functions that other tools are unable to find. Our first RQ considers the effectiveness of our approach *i.e.*, how effective is FUNCRE at recovering library function invocations not recovered by Ghidra.

**RQ1a:** *How effective is our supervised learning-based approach in recovering library function usage?*

In Natural Language Processing (NLP), it is now well-established that pre-training a high-capacity model using self-supervision for an artificial task (*e.g.*, predicting a deleted token, or the following sentence) improves performance on practical tasks like question-answering. Pre-training forces the layers of the model to learn position-dependent embeddings of tokens that efficiently capture the statistics of token co-occurrence in very large corpora. These embeddings are a very useful, transferable representation

of a token and its context [17], [34] which substantially improves performance on other tasks. By using them as an initial embedding within a (possibly different) task-specific network and "fine-tuning" using data labeled specifically for that task, much higher performance can be achieved, even if the task-specific labeled data is limited. The benefits of this approach have also been reported for code [35], [36]. We examine whether pre-training over large corpora of decompiled pseudo-C can be helpful for our task of recovering library function invocations not recovered by Ghidra.

**RQ1b:** *How much does pre-training with ROBERTa help with library function usage recovery?*

C and C++ binaries can be compiled with a variety of optimizations (see Section 2.3). Most disassemblers and decompilers can handle code with no optimizations. In line with that, past research that uses a deep learning (DL) model also targets code compiled with no optimizations. However, in our work, we target higher optimization levels as well. We assess the performance of our model on five optimization levels:

**RQ2:** *How does optimization level affect the performance of* FUNCRE*?*

With machine-learning approaches, the training data can strongly influence the test results. The model might perform better on library functions more prevalent in training data.

**RQ3:** *How does the popularity of library methods influence test results?*

Finally, we assess whether our model outperforms current tools when it comes to retrieving library functions in decompiled code:

**RQ4:** *How does* FUNCRE *perform in relation to state-of-the-art approaches?*

### 3.2 Dataset creation

A key requirement for using supervised machine learning for library function recovery is the creation of a curated, labeled dataset where the occurrence of in-lined functions within decompiled binaries is labeled. There is currently no such dataset of labeled decompiled C/C++ binaries, and we have had to create our own. This presented several challenges.

1) *Large scale, diverse data.* We need a broadly representative, large dataset that captures relevant statistics of current coding styles, library/API usages, compiler settings, and platforms.
2) *Reproducible Builds.* To create binaries with *labeled* inlined library functions we need to suitably instrument the source to insert labels, and then reproduce the build procedures of a large, diverse set of projects. Build procedures are notoriously brittle, with many tricky dependencies, and so challenging to reproduce [37].
3) *Indelible labels.* Because optimizing compilers make substantial changes to the logic of the code, our approach to creating binaries where the original inlined library functions could be labeled in a way that endures after optimization & decompilation is a tricky business.

We employ a multi-stage project selection and build process to meet these challenges (an overview of which can be seen in Figure 3) as elucidated below:

**Large-scale, Diverse data:** The focus of this work is to recover library functions from C-based binaries. Since modern deep-learning models are "data-hungry", we need the largest possible corpus of built binaries aligned with its original C source code. We sourced data from GitHub. Our selection criteria for projects is as follows:

1) *Projects under active development.* We determine a project's activity by checking for commits in the previous six months (as of April 2020). This helps ensure that selected projects are representative of current coding styles and API usages.
2) *Projects using Travis as their CI build platform.* We select those with public build histories and logs (hosted on travis.org) so that we can replicate the builds.
3) *Projects with available Build Containers.* We filter out projects that declare in their Travis configuration (.travis.yml file) that their platform requirement is either Mac OS/X or Linux version 12.04. Travis does not provide open-source Docker containers for either build platform, thus making a build irreproducible.

Our initial selection comprised 10,000 actively developed C-based projects. After filtering for Travis-based projects and then on their build distribution, we are left with 2,634 projects.

**Reproducible Builds:** Successfully re-building projects at scale requires the downloading of each project's dependencies and ensuring that the correct build platform is used. This is a frustrating, failure-prone process under the best of circumstances. These problems are exacerbated when building C projects as there is no standard dependency management system comparable to those in languages such as Java (Maven or Gradle), Python (pip) and, C# (NuGet).

All 2,634 projects in our filtered set of GitHub-based C projects use Travis for continuous integration and require one of three Linux distributions: Trusty (14.04), Xenial (16.04), or Bionic (18.04). Travis CI builds each project in a clean docker container: it first installs all required dependencies and then invokes build and test scripts. We aim to recreate this process.

Fortunately, we were able to leverage the open-source BUGSWARM [37] infrastructure. BUGSWARM was originally developed to reproduce buildable pairs of buggy and fixed versions of large, real-world systems. To ensure reliable builds and reproducible failures, the BUGSWARM pipeline builds pairs of commits and tests them five times. For our purposes, we do not need pairs; we just need reproducible, test-passing (not failing), singleton builds. BUGSWARM is able to identify the right Docker container that a project uses, download the project into the container, install dependencies and build the project using its scripts and the Travis configuration of the project. We only need this part of the pipeline that can build just the latest commit of the code. We downloaded the source-available BUGSWARM [38] project and adapted it for our purposes. First, BUGSWARM currently does not support Travis builds for languages other than Java and Python. We augment BugSwarm's capability to deal with C-based projects. Second, we refactored the BUGSWARM code to retain only a single, buildable version from the latest version of active projects. This adapted version of BUGSWARM called "BUILDSWARM " (which we

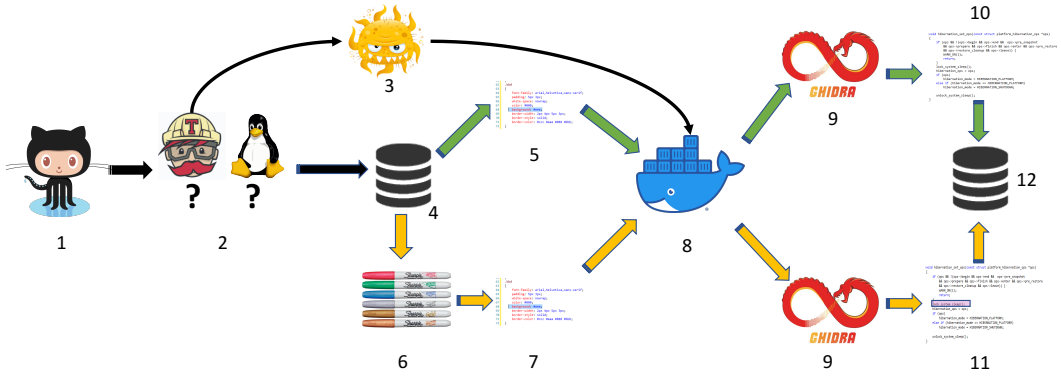


Fig. 3: Training data pipeline. We mine projects from GitHub (1); after filtering (2) the ones enabled for Travis, and certain Operating systems, the projects are gathered in a source dataset (4). We then adapt the publicly available BugSwarm toolset (3) to mine Docker containers (8) for building. We indelibly instrument (6) the library function invocations in the source to get marked source code (7). The raw (5) and marked (7) sources are built using the Docker containers (8); we then use Ghidra (9) to decompile matched pairs (10,11) of marked and unmarked decompiled sources, which are gathered into our labeled dataset (12).

will make available upon publication of this work [39]) works as follows.

1) For each project, we use the Travis API, to download a list of public Travis builds; along with each build, we also download its details, such as build configuration, date of the build, and associated jobs for the build.
2) From this list of builds, we select the latest passing build. Each build might have more than one job associated with it [40] For this build, we select the first job that suits our criteria (1) the job fits our OS criteria (see above), (2) the job does not require Docker as a service (some projects require child Docker containers for testing, a scenario we cannot reproduce) to build the project and (3) the job needs either a `gcc` or `clang` compiler.
3) For the selected job, we create the Travis build script that can replicate the entire build procedure, using the Travis build utility [41].
4) From the downloaded log for the job, we parse the Docker image that was used by Travis. Travis releases most of their docker images on docker hub [42]. We use the same image as our base image to build the project and add the build script to it by composing a docker image.
5) Once this docker image is built, we run the docker build script (generated earlier) on the project inside a docker container. This build script downloads the dependencies builds the code to produce binaries.
6) If the project builds successfully in the container, we release the docker image to Dockerhub [43], and retain that image tag so that the image can be reused; we also collect the pair of the C source file and its object file.

*Disassembling and decompiling a binary* For each project that we can re-build, we need to decompile its binary *i.e.*, convert the executable into a form of pseudo-C code. Section 2.2 explains the process of disassembling and decompiling the binary to recover the pseudo-C code.

The two main tools for disassembling and decompiling a binary are Ghidra and Hexrays IDA Pro. We select Ghidra as our base tool, as it is open source and is freely available; however, we also baseline an evaluation set against both tools, for the specific task of identifying inlined library functions.

Ghidra can disassemble and decompile an executable (.exe file). This entails separating the executable into smaller object files (.o files). Each of these object files is then disassembled by delinking the libraries that they depend on, and then the resulting assembler code is decompiled. In our case, we directly operate on the object files and not on the full executable. This is because we have a one-to-one mapping between the object file and its corresponding C source file. This results in us creating a dataset with source code, binary code, and decompiled code triplets.

**Indelible Labels:** Our machine-learner must learn to associate patterns within the decompiled code with specific inlined library functions. To provide a supervisory signal for the learner, we must *identify* and *locate* a in-lined functions within the Pseudo C code produced by the decompiler; this is non-trivial. It's difficult to align the original C source code with the decompiled Pseudo C, since optimization, followed by the decompilation process, can mangle the original code beyond recognition. Thus, recovering a one-to-one mapping between the original code and the decompiled code is virtually impossible (especially when compiled using higher optimization levels).

To create our dataset of decompiled code with labeled locations for inlined library functions, we need to inject some form of *robust label* into the original C source, that would survive despite optimization transformations and be inserted into the binary; this could then be recovered by a decompiler. We refer to this system of annotating inlined library functions in the binary as *indelible labeling*, or "marking" for short.

The process of marking starts by injecting a marker header in each function in a C project and each function from the libraries used by the project. For our purposes,

we wish to train a learner to learn to identify *just* those functions *not* identified by current decompilers. To find these, we first compile and then decompile source files. For those inlined library functions which are *not recovered* by the decompiler, we must insert a marker that indicates the name of the function and its location within the decompiled Pseudo C code. The marking process must meet several requirements: the injected marker must not interfere with the compilation and decompilation process (no errors should be triggered). Second, there must be no inadvertent changes to the final compiled binary that would differentiate the marked binary from the unmarked: if the resulting decompiled marked Pseudo C differs too much from the original Pseudo C, the training signal for our learner would be confused, and the marked inline library function would not be reliably recoverable by the learner. Third, the injected marker must be resistant to change or removal by the compiler's optimization process. For example, if we were to insert a marker, as a code fragment, whose results were not used elsewhere in the code, for example, something naive like:

```
char *marker1 = "function_inlined: printf()";
```

then the compiler might, for example, find that `marker1` is not used elsewhere, and just simply remove the marking statement, thus robbing us of a data label. We tried several approaches to this problem:

1) `printf`: Injecting a statement that prints the name of the function being inlined. We found that a `printf` statement could at times change the structure of the Ghidra output. For lower optimization levels, the code does not change much; however, for higher optimization levels, the nature of control structures can change *e.g.*, a `while` loop is replaced with a `do while` loop.
2) `puts`: Similar to `printf` the `puts` can print details about the inlined function. While the distortion of the decompiled binary is less than that of `printf`, we do notice that the ordering of statements can be changed, especially for longer function bodies.
3) `Global array marker`: We can inject a global character pointer array (String array) in a source code file and assign it to the array for each function call. Since the array is global, the compiler will not discard it since modifying or discarding such an assignment may change the program semantics.

```
bVar12 = 0;
lVar9 = 0x12;
local_30 = *(ulong*) (in_FS_OFFSET + 0x28);

psVar10 = (sigset_t*) (&local_1f8 + 8);
while (lVar9 != ) {
    lVar9 = lVar9 + -1;
    psVar10->_val[0] = 0;
    psVar10 = (sigset_t *)(psVar10 > _val + 1);
}
```

(a) Without marker

```
bVar12 = 0;
lVar9 = 0x12;
local_30 = *(ulong*) (in_FS_OFFSET + 0x28);
marker21._11 = "function inlined: memset";
psVar10 = (sigset_t*) (&local_1f8 + 8);
while (lVar9 != ) {
    lVar9 = lVar9 + -1;
    psVar10->_val[0] = 0;
    psVar10 = (sigset_t *)(psVar10 > _val + 1);
}
```

(b) With marker

Fig. 4: Marker survives -O2 optimization level without inducing any change in the code

Of these approaches, we chose global arrays to inject markers (Figure 4 depicts one example of an injected marker in decompiled code) in the source code. In comparison to the `printf` and `puts` approaches, the decompiled code obtained from Ghidra is not distorted. This might be due to Ghidra having a harder time in recovering library function calls in the correct position as opposed to array assignment. Furthermore, this tactic ensures that the compiler does not optimize the array access by vectorizing it, which would be the case for linear assignment. The global array we inject is a constant character pointer array of size 2000. We declare a global array and assign each marker to a different position of the array. We note that the *actual value* in the array is not important for labeling; it's the assignment statement itself that constitutes the label.

In each file, we inject a uniquely identified global array, and this helps avoid a compilation conflict. This is necessary because, during compilation, the compiler merges different files (*e.g.*, header files merged into the declaring C file), which might result inadvertently inserting multiple declarations of the same array in one file. For each function call, we assign a marker to a unique position of the array with the name of the function as seen in Figure 4.

If we mark all function calls, we might mark some recovered by Ghidra. Since the learner does not need to recover these, we don't mark them in the code. To remove these markers, for each function definition in the decompiled code, we compare the decompiled function definition bodies with their respective function definition in the original C code. In some cases, function calls from the original C code that are inlined during compilation might be found by the decompiler and indicated as such in the decompiled code. For those function calls that Ghidra recognizes, we remove the marker from the decompiled code; for the rest, we leave the marker as they are an indication of which function call has been inlined and where it has been inlined.

**Identifying target functions:** For this paper, we would like to design an approach that is global *i.e.*, that works on every function that has been inlined. However, for a deep learning-based approach to work, the model has to see one or more examples of an inlined function at training time, allowing it to learn an appropriate representation of each inlined function.

Using our dataset of C projects, we select a set of library functions that could be inlined in the code. We determine the most popular library function calls made by parsing all function calls from the entire dataset of 10,000 projects. To understand which function has been called in a file, we use SrcML [44] to build and resolve the AST and recover the exact function binding.

After parsing all 10,000 projects, we obtain a list of the top 1,000 popularly invoked library functions that we can potentially target. From this 1,000, we filter out those that are never invoked in the 2,634 Travis-based projects, resulting in 724 potential target functions.

### 3.3 Final Dataset Details

We build and obtain binaries from 1,185 (out of 2,634) projects. Many (1,449) projects would not build. For others (726), we cannot find an exact alignment between the source code and the object files. This is because the compiler merges several source files into a single object file, thus confusing file boundaries. In such cases, it is difficult to find the alignment between decompiled pseudo-C, and the original source code, to allow the labeling of the pseudo-C with the requisite inlined functions. However, this decompiled code

is still valuable, and we keep it in our dataset for (RoBERTa) pre-training purposes (as described in the next section).

(a) Cross-file train-test file distribution

| OPT-Level | Train Set | Validation Set | Test Set |
|---|---|---|---|
| O1 | 9 | 0 | 29 |
| Os | 2851 | 88 | 197 |
| O2 | 2710 | 74 | 196 |
| O3 | 2591 | 94 | 144 |
| Of | 482 | 14 | 150 |
| Overall | 8643 | 270 | 716 |

(b) Cross-project train-test file distribution

| OPT-Level | Train Set | Validation Set | Test Set |
|---|---|---|---|
| O1 | 37 | 2 | NA |
| Os | 2840 | 92 | 195 |
| O2 | 2686 | 91 | 207 |
| O3 | 2517 | 96 | 215 |
| Of | 627 | 19 | NA |
| Overall | 8707 | 300 | 617 |

TABLE 1: File-level distribution of the dataset used to train and test FUNCRE

For the other 459 projects, we split the files into training, validation, and test sets in two different settings (file level breakdown presented in Table 1): (1) cross-file where files from the same project can be present in the train, test or validation set and (2) cross-project where all the files from a single project are in one of the train, validation or test sets. In the cross-project setting we do not have enough projects and files for the test set for O1 and Of and thus all evaluation in this setting is done on just three settings. In the cross-file setting, our training set consists of the bodies of 391,967 function definitions spanning 8,643 files and in the cross-project setting we have 401,923 function definitions and 8,707 files. These function bodies are labeled with markers indicating any inlined functions not recovered by Ghidra and used to construct pairs as indicated in Figure 3 in our dataset.

## 4 CREATING FUNCRE

The expected use-case scenario of FUNCRE is shown in Figure 5. To get FUNCRE working, we made several engineering decisions concerning the use of machine-learning techniques. First, we had to select a suitable deep-learning approach. Second, we had to develop an approach to train our models. Finally, we had to design an evaluation methodology to gauge the value added by FUNCRE.

### 4.1 Model Selection

We claim that the task of recovering library function invocations from decompiled pseudo C code resembles text classification in Natural Language Processing (NLP). This intuition's essence: function invocations, especially if inlined, can span multiple lines in decompiled pseudo-C code; some of these lines may contain some pattern that indicates the presence of an invoked library function. We hypothesize that such patterns of pseudo-C code, reflecting the presence of library function invocations, can be learned by a machine learning model given sufficient data and model capacity. In addition, our goal is to *build on top* of the available tools that already recover atleast some function invocations, thus providing greater value to reverse engineers.

**Potential approaches:** We consider two approaches which are the current state of the art in NLP: *Transformers*, and *Masked Language models with fine-tuning*.

*Transformers.* The Transformer [18] model has proven to be very effective for NLP tasks. It is a sequence-to-sequence architecture (*i.e.*, it transforms a given input sequence into another sequence) consisting of an encoder and decoder. The encoder reduces an input sequence into a high dimensional vector, which is fed to the decoder, which outputs another sequence. Older sequence-to-sequence models use a RNN (Recurrent Neural Network) for the encoder and the decoder. Rather than recurrence, Transformers use a multi-head attention architecture along with feed-forward layers. Attention is a mechanism whereby for each token in an input sequence, a model chooses another token therein to "attend to", *viz*, weight in its decision making. A transformer can attend to many tokens in the input sequence, to produce an embedding of an input sequence, using "multi-head" attention, which improves capacity and parallelism beyond RNN (including LSTM and GRU) approaches [18].
*Masked Language Model with fine-tuning.* BERT-based (Bidirectional Encoder Representations from Transformers) [34] masked language models (MLM) leverage self-supervised "pre-training". BERT learns a representation for an input sequence. A BERT model is pre-trained on large unlabeled corpora, using self-supervision, and then fine-tuned for a specific task using standard supervision (*viz.*, explicitly labeled data). This set-up has been shown to outperform traditional Transformer based approaches in NLP. For pre-training, two self-supervised tasks are used: first, it learns to predict masked tokens in the input sequence (typically 15% of the tokens are masked), and second, learning to predict the next sentence following an input sentence (NSP). This pre-trained model is then fine-tuned for a downstream supervised task such as sequence tagging. Currently, it is more common to use ROBERTA (A Robustly Optimized BERT Pretraining Approach) [17] to train a MLM. ROBERTA differs from BERT, with a few changes such as dynamic masking and not doing NSP, but achieves better performance. This setup of pre-training and fine-tuning achieves SOTA results for downstream SE tasks such as variable misuse detection and function-comment mismatch detection [35], [36]. We reuse Huggingface's open-source implementation of ROBERTA [45].

In our setting, labeled training data (for the fine-tuning stage) is somewhat limited, because: (1) we limit the task of function-invocation recovery to only a select set of library functions (see Section 3.2), (2) the number of markers that we are successfully able to insert in the code and recover after the compilation - decompilation loop (see Section 3.2) is limited, (3) only projects that are compiled with a higher optimization level (O2, O3, Os, Of and in rare cases O1) can contain a function invocation that is not already recovered by Ghidra (see Section 2.3). At lower optimization levels, the function calls are not inlined and are more easily found by Ghidra and Hexrays; we don't need to learn patterns for the invocations that are recovered already. As a result of these restrictions, we have a labeled dataset that is smaller than

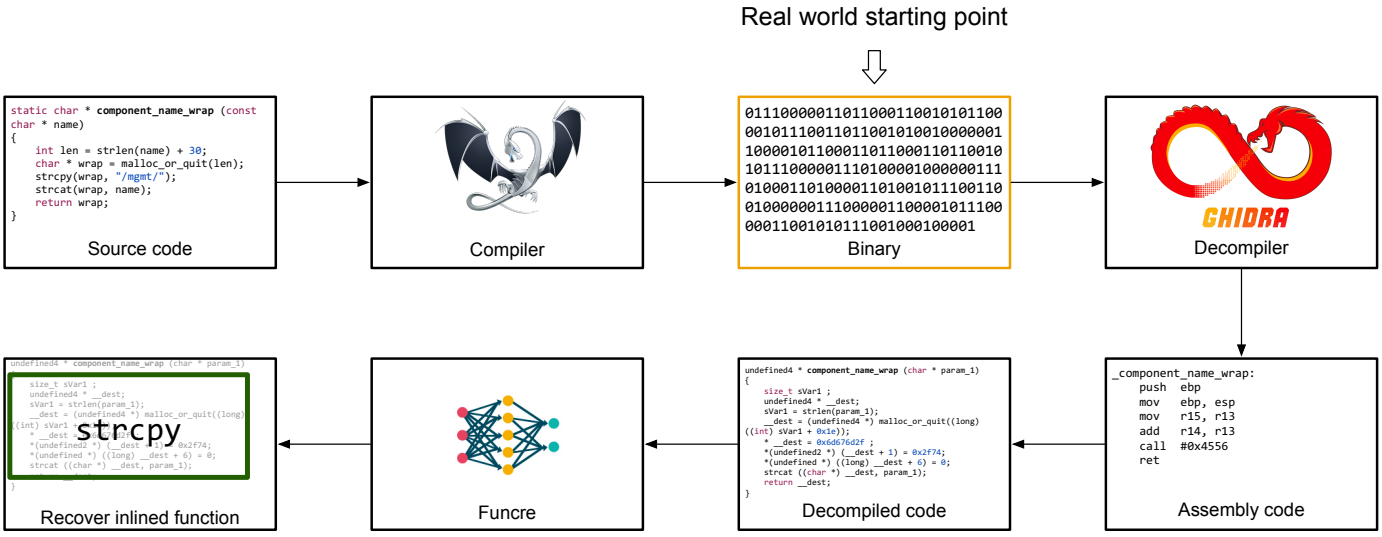


Fig. 5: Working of Funcre. Funcre works on the decompiled output from Ghidra. In a real-world scenario, we start with an external binary. For training and testing purposes we create our own binaries using real-world source code obtained from GitHub.

ideal for these powerful data-hungry transformer models. Thus, the fine-tuning approach (pre-training MLM using ROBERTA and then fine-tuning on our downstream task using the labeled dataset) is well-suited.

For pre-training, we have available a dataset with 1.2B tokens of Pseudo-C files produced by Ghidra, *sans any markers*. We omit markers here to preclude any chances of inadvertent "leaking" knowledge relevant to the final task. Pre-training does learn robust, well-generalized representations of the statistics of decompiled pseudo-C, which enables FUNCRE to quickly patterns that reflect library function invocations, from a few labeled examples. We use the standard pre-training configuration, *viz.*, "ROBERTA base". This configuration has 12 attention layers, 768 hidden dimensions, and 12 self-attention heads in each layer resulting in a model of 125M parameters. We tokenize the code by using a Byte Level BPE (Byte Pair Encoding) Tokenizer. We limit the vocabulary size to 25,000 and keep tokens with a minimum frequency of 20. We train the MLM model on two NVIDIA Titan RTX GPUs for 2.5 epochs with a batch size of 40 sequences. This pre-training process takes three days and achieved a final perplexity of 1.22 when predicting masked tokens. This corresponds to a rather low cross-entropy loss of around 0.36 bits. This suggests that the BERT model is learning a very powerful model of token co-occurrence statistics in the Pseudo-C using the enormous (1.2B token) pretraining data. For comparison, the original ROBERTA paper (for natural language) reported a pre-training final perplexity as low as 3.68 (cross-entropy about 1.8 bits); the significantly higher perplexity for natural language is consistent with prior studies [46].

We end the pre-training once both the training and evaluation loss stops declining further.

*Choosing a Context window size* Before finalizing our model, we needed to address two design issues. (1) We need to determine whether the pre-trained ROBERTA model provides any advantage over simply training a state-of-the-art transformer model directly on task. (2) We need to select a context window-size (in terms of the number of lines the model needs to look at it) that can capture the signature of an in-lined method. Inlined library functions may span multiple lines. If this context window is too narrow, then a pattern capturing a library function invocation (especially if inlined) may not fit in entirely. If it is too big, then it will compromise our ability to locate the method more precisely. Our marking approach indicates the start of the function; however, it does not indicate how many lines it spans. We need to determine what the size of the context window (relative to the start position) must be for a model to effectively learn the representation of a function.

| Context Length | Models | | | | | |
|---|---|---|---|---|---|---|
| | Roberta-base | | | Transformer | | |
| | Top 1 Acc. in % | Top 5 Acc. in % | Top 10 Acc. in % | Top 1 Acc. in % | Top 5 Acc. in % | Top 10 Acc. in % |
| ±3 | 75.73 | 88.33 | 91.44 | 64.93 | 84.56 | 89.28 |
| ±5 | 80.38 | 91.64 | 95.02 | 71.71 | 90.12 | 94.14 |
| +10 | 78.28 | 91.25 | 93.45 | 71.92 | 89.20 | 93.38 |
| −10 | 56.43 | 76.64 | 83.17 | 49.00 | 73.17 | 81.38 |
| ±10 | **80.41** | **92.56** | **95.21** | 69.37 | 89.41 | 93.13 |

TABLE 2: Performance of RoBERTa and Transformer models on development set at different context size

**Finalizing our Design:** We evaluated our design choices on our validation set, using a straw-man task. We train a vanilla Transformer model end-to-end on the labeled training dataset. For the ROBERTA-based model, we reuse the pre-trained MLM mentioned earlier and then fine-tune it on our labeled dataset. We train the models by feeding them input sequences where the function invocation marker is in the middle surrounded by a context window (in terms of the number of lines) of varying sizes. This form of training does not parallel a real-world setting where the location of function invocation is typically unknown; however, to choose a model architecture, this approach is reasonable.

In Table 2 we see the performance of both the ROBERTA model and the Transformer model on five different context

window sizes. We observe that the window size $\pm 10$ works best in the case of ROBERTA. We also notice that the context window sizes $+10$ and $-10$ do not work as well as a context window that spans both sides of a marker. This implies that the model requires both the preceding and succeeding lines of code to learn the representation of a function invocation. Both ROBERTA and the Transformer model learn a fairly good representation of the invocations. With the Top 1 accuracy for ROBERTA reaching 80% and the Top 5 accuracy being 92.5% (both in the case of context window size $\pm 10$). Furthermore, we see that in Table 2 the ROBERTA model outperforms the Transformer model in every setting. This implies that by pre-training a MLM and then fine-tuning it on the task, we can achieve better performance.

*We, therefore, chose the MLM architecture, and* $\pm 10$ *context window size, for our more realistic training/evaluation regime.*

### 4.2 Final Training & Evaluation

Using the pre-trained MLM, now we fine-tune the MLM using the labeled dataset to create FUNCRE. In the earlier straw-man approach, which was used model selection, we assumed that the model would know the location of the library function invocation. In a real-world setting, the model must recover the invocation from the entirety of the decompiled code for a function definition, without knowledge of the specific location. Because of optimizations such as inlining and code movement, it's often not possible to know exactly where in the pseudo-C the function invocation should be located; we therefore relax the problem to locating the *calling function body* in which the invocation occurs. A correct recovery for us would therefore amount to recovering the correct invoked method, and the correct function body in which that method is invoked. Our recovery works by scanning a window over the a function body and looking for invocations within each window.

**Scanning the Window:** Based on the indication from the straw-man evaluation above, we employ a context window of $\pm 10$ size to both train and test our model. For function bodies that are over 20 lines long, we slide a context window forwards, one line at a time. In both training and test, each sliding window is labeled with the marker that occurs in that window. When there are multiple markers in a window, we simplify the labeling for the block by marking it with the first marker in lexical order. This line-by-line scanning does present a problem. Consider an inlined library function invocation (say `atoi`) that occurs at line 8 of a 30-line function. The corresponding marker will occur around line 8 in the first 20-line scanning window (starting at line 1 of the function) and repeat eight or more times as the 20-line scanning window moves forward, a line at a time. The learner may find an invoked function's signature in even more than eight successive windows. Consequently, we adopt a sequential filtering heuristic to coalesce these repeating sequential labels, as described next.

**Coalescing Sequential Labels:** We use a simple, noise-tolerant run-length encoding heuristic to coalesce sequential blocks. This heuristic was tuned on the validation set without examining the test set.

1) Given a sequence of predicted labels in long function, we remove sequentially inconsistent predictions, *viz*, labels that disagree with the five preceding *and* succeeding labels. This works well since in-lining is rare in shorter functions.
2) We use run-length encoding on the sequence. Run-lengths are incremented leniently; our heuristic will treat up to 3 successive unlabeled windows as being the same as preceding label and succeeding label (if both preceding label and succeeding label are same). Thus if $a, b, c$ are method labels and $x$ is a "no method" label, the label sequence $aaxxaabbbbbxcxxxxcdc$ is encoded as $a^6b^5c^1c^3$. The second two $x$ after the first two $a$ are treated as $a$, so we get a total run length of 6 $a$; after the 5 $b$, the $c$ is accepted, although it is inconsistent with the preceding $b$ because it agrees with a following $c$; the $d$ within the run of 2 $c$ at the end is erased for being inconsistent.
3) Next, we only retain function labels with a run-length of at least four as a true label. The above example then collapse to just $a^6b^5$. Finally, we divide the run-length by 20 and take the ceiling. This leaves us with just two labels, $a$ and a following $b$. We collect the markers from the Ghidra output and compare the result. Finally, we add the result to the result achieved by Ghidra.

We note again that this heuristic was tuned **exclusively** *on the validation set*, to scrupulously avoid overfitting to the test set.

**Data Imbalance:** After dividing both the training and validation sets into windows containing 20 lines of code shifted by one line, we observe that our dataset is imbalanced: the unlabeled windows dominate. Since we have a pre-trained ROBERTA model that has learned the statistics of the unlabeled window, we re-balance the data by discarding some 65% of these blocks with no label. Even so, the no-label windows are about 80% of our training and development set. For the fine-tuning stage, we employ the same tokenizer that was used for pre-training. We fine-tune our model over three epochs on six NVIDIA Titan RTX GPUs, taking a total of three hours.

## 5 EMPIRICAL RESULTS

Our goal is to improve the performance of Ghidra in recovering inlined library functions. Ghidra already recovers some library functions; the combination of Ghidra with our model *should* improve performance. We begin with our evaluation metric and then dive into our results.

**Evaluation Metric:**

We remind the reader (as described in Section 4.2 ) a correct recovery for us is a library function invocation, together with the function body in which this invocation occurs. Thus *a single test instance is a specific library function invocation, together with the containing function; this is our target for recovery*. This task is performed by scanning candidate function bodies in 20-line blocks; we explain above Section 4.2 how the model decides if and what function invocations occur in these blocks. In the following we explain our evaluation criteria as it applies to the problem of *recovering library function invocations within function bodies*, *viz.* how we decide if a test instance results in a TP, FP, TN, FN, *etc*

- *Model predicts empty (*i.e.*, no invoked function) and true label is empty: this function body contains no library function*

*invocations* we mark this as a true negative (TN). These are of limited value to the RE and extremely numerous! So we ignore these in our calculations (note that we do not report "accuracy", and do not count these in our precision calculation).
- *Model predicts 'Func1' and true label is 'Func1':* we count it as a true positive (TP). These are helpful to the RE.
- *Model predicts label 'Func1', and true label is empty:* we count a false positive (FP). In this case, our model is confused, and finds a library function invocation in the body of another function, where there isn't any such invocation. These create needless work for the RE.
- *Model predicts empty, and true label is 'Func1'* we count a false negative (FN). Our model failed to recover a library function invocation that did occur within a function body. These cases fail to provide useful information to the RE.
- *Model predicts label 'Func1' and true label 'Func2:* we score this as a false negative, FN, (for missing 'Func2'), and also a FP (for predicting 'Func1'). Our model not only failed to recover a function invocation, it also reported incorrectly that a different function was used than the actual one! These cases fail to report important information, *and* create needless work, and so is doubly penalized.

As can be seen from the above, TP+FN is the count of actual function invocations that *could* be recovered. Based on these counts for FP, FN, TP, and TN, we calculate the precision, recall, and F-score that our model achieves on the test set. Note again that we ignore the correctly labeled empties despite this being an instance of our model performing well (this lowers our precision from ∼0.90 to ∼0.60) since it is of limited value to the RE. All the evaluations are given below use these criteria.

### 5.1 RQ1: Effectiveness of FUNCRE

(a) Cross-file train-test split

| OPT-Level | TP | FP | FN | Prec. | Recall | F-score |
|---|---|---|---|---|---|---|
| O1 | 135 | 86 | 130 | 0.61 | 0.51 | 0.56 |
| Os | 752 | 366 | 867 | 0.67 | 0.46 | 0.55 |
| O2 | 647 | 403 | 760 | 0.62 | 0.46 | 0.53 |
| O3 | 736 | 437 | 955 | 0.63 | 0.44 | 0.51 |
| Of | 898 | 429 | 998 | 0.67 | 0.47 | 0.56 |
| Overall | 3168 | 1721 | 3710 | 0.64 | 0.46 | 0.54 |

(b) Cross-project train-test split

| OPT-Level | TP | FP | FN | Prec. | Recall | F-score |
|---|---|---|---|---|---|---|
| O2 | 1004 | 417 | 923 | 0.70 | 0.52 | 0.60 |
| O3 | 743 | 1192 | 1222 | 0.38 | 0.38 | 0.38 |
| Os | 927 | 641 | 919 | 0.59 | 0.50 | 0.52 |
| Overrall | 2674 | 2250 | 3064 | 0.54 | 0.46 | 0.50 |

TABLE 3: Performance of FUNCRE at various optimization levels

Now, we start with the results for FUNCRE on the task of recovering library function invocations *not* recovered by Ghidra (we evaluate the recovery of *all* function invocations later). Our evaluation is based on two different dataset splits as elucidated in Section 3.3. Table 3a presents our model's performance on the test set that has been split at file level. Overall test instances, FUNCRE achieves a precision of 0.64 and a recall of 0.46. We have more FNs than FPs, suggesting that the model errs on the side of caution *i.e.*, instead of inaccurately predicting the presence of a function, the model predicts that there is no function in place.

In Table 3b we present our results on a test that has been split at project level. With this test split we only have sufficient training and test data for three optimization levels - O2, O3, Os as for O1 and Of the data originates from a few projects thus rendering a project level split impossible. We observe that with this cross project split, the precision and recall at O2 increases in comparison with the cross-file split. We see that for Os we lose some precision but improve the recall and overall the F-score is slightly lower. However, for O3 we see a degradation in both precision and recall.

Since the data in our cross-file dataset is so imbalanced, we check if our model performs better than a random model or coin toss. We made several runs of a simulated model that uses only prior probability to predict invoked function. Not surprisingly, FUNCRE vastly outperforms a simulated model that guesses labels just based on priors: the f-score never rises above 0.003 despite hundreds of simulated runs.

Figure 6 shows three code snippets from our test set which contain the functions `memset`, `strcmp`, and `die` respectively. Despite the lack of obvious signs in the decompiled code of these invoked functions, our model can identify the functions correctly. This suggests that our model is learning a useful representation of the decompiled code and can recover (even inlined) function calls. Table 4 presents a sample list of library functions recovered by Funcre. Several of these represent vulnerabilities, and/or malicious behavior; in general, labeling unidentified library function calls correctly, in decompiled code, represents useful information that is *simply not* otherwise available to the reverse engineer.

```
memset, fprintf, check, setjmp, match, snprintf, free,
fopen, wifexited, closesocket, xmalloc, htons, calloc,
testnext, malloc, open, localtime, wifsignaled, impossible,
fail, unlock, xstrdup, pixgetdata, verbose, validate, typeof,
getpid, strcasecmp, warnx, waitforsingleobject, getgid,
system, entercriticalsection, createevent, setsockopt, raise
crc32, leavecriticalsection, perror, chmod, report
```

TABLE 4: Functions recovered by FUNCRE.

**Finding 1**. Overall, the code representation learned by FUNCRE is powerful enough to recover 46% of library function calls, and errs on the side of caution (more FN than FP)

### 5.2 RQ2: Effect of Optimization Level

We look at the effect of optimization level on FUNCRE in only the cross-file setting as the cross-project setting has not been evaluated on two optimization levels. In Table 3a we examine model performance at varying optimization levels: higher optimization levels make the task harder. For the O1 level, we have reduced training data due to the low rate of function inlining that occurs; thus we see the poorest performance. Likewise, we have reduced training data for Of; however, we see that model's precision is higher in this case. This despite Of being the most complex optimization; we hypothesize that is because of inductive transference

```
char * local_28 ;
long local_20 ;
local_20 = * (long *) (in_FS_OFFSET + 0x28);
local_28 = (char *) 0x0;
if ((*param_1 == '-') && (param_1 [1] == '\0')){
    uVar4 = esl_buffer_OpenStream(stdin, param_3);
    goto LAB_0010192f;
}
iVar2 = esl_FileExists(param_1);
if (iVar2 == 0) {
uVar3 = esl_FileEnvOpen(param_1, param_2, 0, &local_28);
uVar4 = (ulong) uVar3;
```

(a) Inlined function: `strcmp`

```
    uVar5 = 0xffffffff;
} else {
    local_30 = 0;
    local_38 = 0;
    __memcpy_chk((long) &local_38 + 4, *phVar2 ->
            h_addr_list, (long) phVar2->h_length, 0xc);
    local_38 = CONCAT62 (CONCAT42(local_38._4_4_,
            param_2 >> 8 | param_2 << 8), 2);
    iVar1 = connect(__fd, (sockaddr*) &local_38, 0x10);
    if (iVar1 < 0) {
        rl_fprintf(stderr, "ERROR connect");
        piVar3 = __errno_location();
        pcVar4 = strerror(*piVar3);
```

(b) Inlined function: `memset`

```
uint __fd;
int iVar1;
uint* puVar2;
char* pcVar3;
char* __format;
FILE* __stream;
__fd = kqueue();
if ((int) __fd < 0) {
    __stream = *ppFRam0000000000100301;
    puVar2 = (uint*) __errno_location();
    pcVar3 = strerror (*puVar2);
    __fd = *puVar2;
    __format = "%s(): kqueue(): %s (%i)\n";
} else {
    _test_no_kevents((ulong) __fd, "/home/travis/build
        test/main.c", 0x97);
```

(c) Inlined function: `die`

Fig. 6: Example code snippet from decompiled code containing an inlined library function

from the O3, O2, and Os, classes, where many similar optimizations may occur.

For Os, O2, and O3, the F-score ranges between 0.55 and 0.51. Additionally, in all three cases, the precision is higher than 0.60. As the optimization level increases, the precision drops slightly. However, the recall remains almost constant. This relatively stable performance suggests that the model can deal with the challenges posed by higher optimizations, where the decompiler typically struggles to create a helpful representation.

**Finding 2**. The performance of FUNCRE does not deteriorate significantly with the increasing complexity of compiler optimization.

### 5.3 RQ3: Impact of the popularity of methods

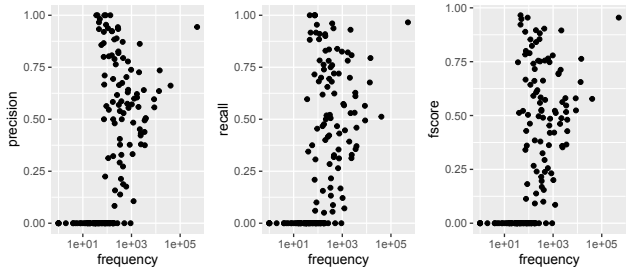


Fig. 7: Does training set frequency affect model performance?

How does performance vary with training sample frequency? Figure 7 plots the method frequency in the training set against precision, recall, and F-score in the cross-file setting (we omit the cross project setting here due to the relative lack of diversity of the test set). We can see that for methods that occur less than roughly 50 times in the training dataset, performance is generally quite low. These methods include `strtoi` (frequency is 1), `vprintf` (frequency is 20) and `rand` (frequency is 30). At intermediate frequencies, between 50 and 1500, performance is quite variable. There are some popular methods such as `offsetof` (frequency is 917) and `max` (frequency is 1,220) for whom the F-score remains quite low, near 0. However FUNCRE can perform well on functions such as `lseek` (frequency is 90) and `strndup` (frequency is 72) where the F-score is higher than 0.8. We conjecture that performance depends on other factors, *e.g.*, how varied the invocation (or inlined) code looks for each function.

At much higher frequencies, performance more reliably improves; methods such as `sscanf` (frequency is 2,186), `printf` (frequency is 14,437) and `assert` (frequency is 40,520) all show good performance. FUNCRE is also able to perform well on rares functions such as `lseek` (frequency is 90) and `strndup` (frequency is 72) where the F-score is higher than 0.8.

The overall Pearson correlation values of the frequency with precision is 0.14, with recall 0.16, and with F-score 0.17.

**Finding 3**. The performance of FUNCRE has a weak correlation with call frequency. The weak correlation arises from 3 distinct regions of performance: consistently poor performance at low frequencies, very variable in mid-ranges, and more reliably higher at higher frequencies.

### 5.4 RQ4: Comparison to existing tools

(a) Cross-file train-test split

| OPT. level | File Count | Total Functions | Tool | TP | FP | FN | Prec. | Recall | F-score | Unique Function Recovered |
|---|---|---|---|---|---|---|---|---|---|---|
| O1 | 29 | 867 | Hex-Rays | 456 | 103 | 407 | 0.81 | 0.53 | 0.64 | 50 |
| | | | Ghidra | 460 | 67 | 404 | **0.87** | 0.53 | 0.66 | 51 |
| | | | Ghidra+FUNCRE | 595 | 153 | 400 | 0.80 | **0.59** | **0.68** | **54** |
| Os | 197 | 2932 | Hex-Rays | 1796 | 1819 | 2665 | 0.50 | 0.40 | 0.44 | 100 |
| | | | Ghidra | 1632 | 1399 | 2983 | 0.54 | 0.36 | 0.43 | 89 |
| | | | Ghidra+FUNCRE | 2384 | 1765 | 3144 | **0.57** | **0.43** | **0.49** | **108** |
| O2 | 196 | 1686 | Hex-Rays | 2234 | 1325 | 2764 | 0.62 | 0.44 | 0.52 | 131 |
| | | | Ghidra | 2012 | 934 | 3129 | **0.68** | 0.39 | 0.50 | 126 |
| | | | Ghidra+FUNCRE | 2659 | 1337 | 3256 | 0.67 | **0.45** | **0.54** | **160** |
| O3 | 144 | 1747 | Hex-Rays | 2499 | 3330 | 3279 | 0.43 | 0.43 | 0.43 | 104 |
| | | | Ghidra | 2334 | 2191 | 3699 | 0.52 | 0.39 | 0.44 | 98 |
| | | | Ghidra+FUNCRE | 3070 | 2628 | 3950 | **0.53** | **0.43** | **0.48** | **126** |
| Of | 150 | 1579 | Hex-Rays | 1111 | 489 | 2840 | **0.69** | 0.28 | 0.40 | 78 |
| | | | Ghidra | 893 | 656 | 3282 | 0.58 | 0.21 | 0.31 | 75 |
| | | | Ghidra+FUNCRE | 1791 | 1085 | 3411 | 0.62 | **0.34** | **0.44** | **89** |

(b) Cross-project train-test split

| OPT. level | File Count | Total Functions | Tool | TP | FP | FN | Prec. | Recall | F-score | Unique Function Recovered |
|---|---|---|---|---|---|---|---|---|---|---|
| O2 | 207 | 1250 | IDA | 2028 | 2028 | 3917 | 0.34 | 0.27 | 0.30 | 110 |
| | | | Ghidra | 1864 | 1593 | 3664 | 0.53 | 0.33 | 0.41 | 105 |
| | | | Ghidra+Us | 2868 | 2010 | 3633 | **0.58** | **0.44** | **0.50** | **123** |
| O3 | 215 | 2496 | IDA | 1784 | 5031 | 5323 | 0.26 | 0.25 | 0.26 | 115 |
| | | | Ghidra | 1623 | 2030 | 3771 | **0.44** | 0.30 | 0.36 | 110 |
| | | | Ghidra+Us | 2366 | 3222 | 4346 | 0.42 | 0.35 | **0.39** | **134** |
| Os | 195 | 2687 | IDA | 2498 | 3967 | 5362 | 0.39 | 0.32 | 0.35 | 102 |
| | | | Ghidra | 2363 | 1280 | 3115 | **0.64** | 0.43 | 0.52 | 98 |
| | | | Ghidra+Us | 3290 | 1921 | 3242 | 0.63 | 0.50 | **0.56** | **113** |

TABLE 5: Comparison of Ghidra + FUNCRE, Ghidra and Hexrays

Next, we compare our overall performance to the state-of-the-art, Ghidra, and Hex-rays. In this evaluation, we

consider recovery of *all* function invocations. How well do the available tools identify function invocations, whether inlined or not? Since FUNCRE works on top of Ghidra, we augment Ghidra's results with ours to measure if (and how much) the function recovery of Ghidra is improved.

To assess the TP and FP rates for function recovery with Ghidra and Hex-rays, we compare the decompiled code against the original source code to see how many functions are recovered correctly. We run this analysis on our test set in both the cross-file and cross-project setting. Compared to Table 3a, we are missing the results for Hex-rays on 14 files in our test set, because the version of Hex-rays at our disposal cannot process 32-bit binaries[3], so we just omitted them from this comparison, to be generous to Hex-rays.

In Table 5a we see that Ghidra + FUNCRE has the best f-score for all optimizations (precision declines somewhat for O1 and Of, and marginally for O2). Our tool does not degrade the recall or F-score for Ghidra; instead, it enhances it enough to outperform Hex-rays. We also see that Ghidra + FUNCRE recovers the most library function calls, and in all cases, also the most *unique* functions: *e.g.*, at the O2 level, Ghidra + FUNCRE recovers 29 more unique functions than Hex-rays, while achieving also getting the highest F-score.

All outputs of FUNCRE, Hex-ray, and Ghidra are multisets. If one `EnterCriticalSection` and 3 `sprintf` are actually inlined, Ghidra may recover partially (e.g, one `sprintf` is recovered), and we combine those with our output. To combine outputs, we take a multiset union (which could boost both TP and FP, and reduce FN; note that both function name and count matter to measure Recall/Precision/F1). Table 3a reports on JUST the MARKED functions (we evaluate on the recovery of 2 sprintf and 1 EnterCriticalSection) recovered per optimization level by Funcre ALONE.

We repeat the same analysis in our cross project setting as well and present the results in Table 5b. We see that in contrast to the cross file setting, the precision and recall is down in all three scenarios across all three optimization levels. Despite this downturn, Ghidra + FUNCRE outperforms plain Ghidra and Hex Rays in terms of F-score in all the cases. We do notice that the precision and recall for O2 and O3 are lower than in the cross file setting, however, for Os there is an increase. Overall, in the cross project setting we see a drop in performance in comparison to the cross file setting, but even in this setting FUNCRE shows that it outperforms the competition.

We find it noteworthy that the Hex-ray's FP count is this high given that the Hex-rays developers explicitly designed their FLIRT signature system to be cautious and never introduced a false positive (see Section 2.3).

To investigate further, we examine a random subset of 10 function definitions containing one or more FP library calls for each of the three tools. We observe that in the majority (seven for Hex-rays, seven for Ghidra, and five out of ten for Ghidra + FUNCRE) of the cases, the FP are correctly marked as FP *i.e.*, the tool incorrectly recovers the wrong function based on a comparison with the original source. In the remaining cases, we find that the function call is transitively inlined from a function definition or macro from another file and due to the limitation in our detection strategy (these cases are near impossible to detect due to the absence of a system-level call graph) are marked as FP.

Out of 724 popularly used library functions present in the projects we target , only 365 are inlined depending on optimization level (O1 has minimal inlining). Only 168 occur in our test set, and Funcre recovers 93. Significantly, we improve 19% over Ghidra and 10% over Hex-rays; for the challenging case of O2, we improve by 22% over Hex-rays (second highest in Table 5a). Note that we correctly recover more inlined-functions; our recall improves over Hex-rays e.g., for Of by over 20%, finding 680 more instances of inlined-functions. While improving recall, Funcre leaves precision about the same or improved (Table 5a). Our FP rate *is not higher* than current tooling.

**Finding 4**. FUNCRE enhances the performance of Ghidra to the extent that it outperforms Hex-rays.

## 6 THREATS TO VALIDITY

**Generalizability.** We collect code and build binaries from GitHub projects; these may not yield binaries that are typically reverse-engineered. Furthermore, we only target binaries built against three versions of Linux on an x86 architecture; performance on other platforms may vary.

**Internal validity.** We mark predictions as true only when exactly matched with the expected label. However, in some cases, the decompilers recover functions such as `printf_chk` and `assert_fail` rather than `printf` and `assert`. One could argue that the prediction is correct in such cases, but we still mark it as incorrect. This impacts measured scores equally for all tools (FUNCRE, Ghidra, and Hex-rays). This is a non-trivial issue with no easy resolution: *e.g.*, not requiring an exact match also risks biases & errors.

Certain false positives in the function recovery for all three tools also originate from the fact that function definitions or macros from the same project might have been inlined into the function that we analyze. Inlined calls transitively inlined in the function under consideration might be recovered by all three tools; however, we have no way of knowing whether the recovery is a true positive. We do look at this transitive inlining for one step *i.e.*, for the first function declaration inside the same file that is inlined. However, we do not construct a system-level call graph which might adversely impact the false positive rate reported for all three tools.

**Practical applicability.** When both Ghidra and Hex-rays recover a function call, they can place it in the function definition body as an actual function call and not a collection of statements. FUNCRE can only recover the list of inlined functions per function declaration body. However, as seen in Section 5.4 we observe that both Ghidra and Hex-rays have false positives when it comes to function recovery, furthermore, the evaluation strategy employed in this work does not know whether the location where the function is recovered is correct or whether the parameters that are passed to the function call are correct. We do recover some functions that current tools cannot; still, marking the exact

3. A commercial license for the 32 bit version of Hexrays is available for additional purchase

position of the inlined is function is much harder because the compilation-decompilation loop can move the location of a marker. Knowing exactly which lines correspond to an inlined function is also non-trivial without a more advanced representation of the code.

## 7 Contributions

We have described an approach to improve inlined library function recovery in comparison with state-of-the-art tools (Ghidra and Hex-rays IDA Pro). Our main contributions are:

1) We created a technique to build C-based projects on a large scale. Using this pipeline, we build and release the Docker containers for 1,185 C projects. We also created an annotated set of real projects (first of its kind), which indicates functions inlined by compilers. Our data & tooling will be released.
2) We show that MLM pre-training, on 1.2 billion tokens of decompiled code improves task-performance for inlined library function recovery. We will release the trained MLM for reuse in other RE tasks, such as name recovery.
3) We improve upon Ghidra and Hex-rays on library function recovery, both in terms of f-score and unique functions recovered. This suggests that modern machine-learning methods have value for this task.
4) There has been less attention in prior research work (on binary analysis) towards highly optimized binaries. Our work considers all optimization settings, including the highest (Of). This suggests that our research has greater relevance in broader settings than prior work.

## Acknowledgments

We gratefully acknowledge support from NSF CISE (SHF LARGE) Grant No. 1414172, and from Sandia National Laboratories. Toufique Ahmed is supported by a Dean's Distinguished Graduate Fellowship. This paper was much improved as a result of reviewers' comments, for which we are thankful.

## References

[1] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.
[2] H. B. BE, "Hexrays ida pro," http://hex-rays.com/products/ida/, last Accessed August 2020.
[3] NSA, "Ghidra," https://ghidra-sre.org/, last Accessed August 2020.
[4] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2000, pp. 38–49.
[5] T.-Y. Wang, S.-J. Horng, M.-Y. Su, C.-H. Wu, P.-C. Wang, and W.-Z. Su, "A surveillance spyware detection system based on data mining methods," in *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 2006, pp. 3236–3241.
[6] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: Intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 1043–1047.
[7] Y. Ye, T. Li, Q. Jiang, and Y. Wang, "Cimds: adapting postprocessing techniques of associative classification for malware detection," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 3, pp. 298–307, 2010.
[8] Y. Ye, T. Li, K. Huang, Q. Jiang, and Y. Chen, "Hierarchical associative classifier (hac) for malware detection from the large and imbalanced gray list," *Journal of Intelligent Information Systems*, vol. 35, no. 1, pp. 1–20, 2010.
[9] Y. Ye, T. Li, Q. Jiang, Z. Han, and L. Wan, "Intelligent file scoring system for malware detection from the gray list," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 1385–1394.
[10] M. M. Masud, J. Gao, L. Khan, J. Han, and B. Thuraisingham, "Mining concept-drifting data stream to detect peer to peer botnet traffic," *Univ. of Texas at Dallas, Tech. Report# UTDCS-05-08*, 2008.
[11] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," in *2010 5th international conference on malicious and unwanted software*. Ieee, 2010, pp. 23–30.
[12] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 646–656, 2013.
[13] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–40, 2017.
[14] H. B. BE, "Flirt signatures," https://www.hex-rays.com/products/ida/tech/flirt/in_depth/, last Accessed August 12th 2020.
[15] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 261–270.
[16] H. B. BE, "Hexrays ida pro inlined function recovery," https://hex-rays.com/products/decompiler/compare/v12_vs_v11/, last Accessed August 2020.
[17] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[19] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. IEEE, 2001, pp. 602–611.
[20] G. Hunt and D. Brubacher, "Detours: Binaryinterception ofwin 3 2 functions," in *3rd usenix windows nt symposium*, 1999.
[21] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.
[22] U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A tool for analyzing malware*. na, 2006.
[23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.
[24] J. Qiu, X. Su, and P. Ma, "Using reduced execution flow graph to identify library functions in binary code," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 187–202, 2015.
[25] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 301–324.
[26] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.
[27] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.
[28] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "{BYTEWEIGHT}: Learning to recognize functions in binary code," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 845–860.
[29] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *2017 IEEE International*

*Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 388–398.
[30] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 611–626.
[31] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.
[32] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *arXiv preprint arXiv:1905.08325*, 2019.
[33] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.
[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
[36] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.
[37] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.
[38] BugSwarm, "Bugswarm githubory repository," https://github.com/BugSwarm/bugswarm, last Accessed August 2020.
[39] Blinded, "Replication package for this work," https://doi.org/10.5281/zenodo.4007527, last Accessed August 2020.
[40] Docker, "Docker job matrix configuration," https://docs.travis-ci.com/user/build-matrix, last Accessed August 2020.
[41] T. CI, "Travis build utility," https://github.com/travis-ci/travis-build, last Accessed August 2020.
[42] ——, "Travis dockerhub repository," https://hub.docker.com/u/travisci, last Accessed August 2020.
[43] Blinded, "Docker containers created," https://hub.docker.com/r/binswarm/cbuilds/tags, last Accessed August 2020.
[44] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcml toolkit," in *2011 IEEE 11th international working conference on source code analysis and manipulation*. IEEE, 2011, pp. 173–184.
[45] Huggingface, "Huggingface transformers," https://github.com/huggingface/transformers, last Accessed August 2020.
[46] C. Casalnuovo, K. Sagae, and P. Devanbu, "Studying the difference between natural and programming language corpora," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1823–1868, 2019.

**Toufique Ahmed** is a Ph.D. student at UC Davis. He received his B.Sc. and M.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) in 2014 and 2016. His research interest includes Software Engineering, the Naturalness of Software, Machine Learning, and Sentiment Analysis. He is the recipient of the five-year prestigious Dean's Distinguished Graduate Fellowship (DDGF) offered by The Office of graduate studies, The College of Engineering, and The Graduate Group in Computer Science, UC Davis.

**Premkumar Devanbu** received his B. Tech from IIT Madras and his Ph.D from Rutgers University. He is currently Distinguished Professor of Computer Science at UC Davis. His research interests include Empirical Software Engineering and the applications of the Naturalness of Software, including Machine Learning applied to Software Engineering.

**Anand Ashok Sawant** is a Postdoctoral scholar working on the application of Machine Learning for Software Engineering at the Decal lab at the University of California Davis, USA. He got is PhD at the Delft University of Technology in 2019, where his research was focused on studying API evolution.