Received 4 January 2024, accepted 11 January 2024, date of publication 17 January 2024,
date of current version 26 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3355098

RESEARCH ARTICLE

ToolPhet: Inference of Compiler Provenance From Stripped Binaries With Emerging Compilation Toolchains

HOHYEON JANG, NOZIMA MURODOVA, AND HYUNGJOON KOO
Department of Computer Science and Engineering, Sungkyunkwan University, Suwon-si, Gyeonggi-do 16419, Republic of Korea

Corresponding author: Hyungjoon Koo (kevin.koo@skku.edu)

This work was supported in part by the Basic Science Research Program through National Research Foundation of Korea (NRF) Grant funded by the Ministry of Education, Government of South Korea, under Grant 2022R1F1A1074373; and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government [Minister of Science, Information and Communications Technology (MSIT)], Graduate School of Convergence Security, Sungkyunkwan University, under Grant 2022-0-01199.

ABSTRACT Identifying compiler toolchain provenance serves as a basis for both benign and malicious binary analyses. A wealth of prior studies mostly focuses on the inference of a popular compiler toolchain for C and C++ languages from stripped binaries that are built with GCC or clang. Lately, the popularity of an emerging compiler is on the rise such as Rust, Go, and Nim programming languages that complement the downsides of C and C++ (*e.g.*, security), which little has been explored on them. The main challenge arises when applying previous inference techniques for toolchain provenance because some emerging compilation toolchains adopt the same backend of traditional compilers. In this paper, we propose ToolPhet, an effective end-to-end BERT-based system for deducing the provenance of both traditional and emerging compiler toolchains. To this end, we thoroughly study the characteristics of both an emerging toolchain and an executable binary that is generated by that toolchain. We introduce two separate downstream tasks for the compiler toolchain inference with a (BERT-based) fine-tuning process, which produces 1) a toolchain classification model; and 2) a binary code similarity detection model. Our findings show that the classification model 1) may not suffice when producing a binary with the existing backend like Nim, which we adopt the detection model 2) that can infer underlying code semantics. We evaluate ToolPhet with the previous work including one signature-based tool and four machine-learning-based approaches, demonstrating its effectiveness by achieving higher F1 scores with the binaries compiled with emerging compilation toolchains.

INDEX TERMS Compiler inference, binary analysis, BERT, classification model, similarity model.

I. INTRODUCTION

Software is everywhere, ranging from personal laptops, cloud services, mobile phones, IoT (Internet of Things) devices to artificial satellites. However, understanding the internals of software is notoriously challenging because an executable binary as a distribution form has been stripped high-level information. Reverse engineering (reversing) of the binary assists in deducing its underlying semantics with varying tasks including function boundary identification [1], [2], [3], compilation toolchain provenance [4], [5], [6], binary similarity detection [7], [8], known vulnerability detection [9], class hierarchy inference [10], and function and variable symbol name prediction [11], [12].

An executable binary is produced by a compiler toolchain. However, different settings (*e.g.*, optimization levels, versions, flags) of the same toolchains as well as different compiler families can exhibit different binaries although the semantics of binary code are identical. Recognizing a compiler toolchain for binary reversing serves as a basis for

The associate editor coordinating the review of this manuscript and approving it for publication was Mostafa M. Fouda.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

(both benign and malicious) binary analysis from a digital forensic viewpoint. A good utility would be investigating one of the malware characteristics. For example, malware variants that are distributed via the black market (*e.g.*, Malware-as-a-service) may have the lineage by the original malware authors. Moreover, identifying a compiler toolchain can be helpful in understanding the specifics of a binary; *e.g.*, a compiler has its own implementation for checking a canary or control flow integrity.

The most popular compiler toolchains are GCC and clang that support C and C++ due to their benefits of speed, portability, flexibility, efficiency, and well-defined standard libraries. Despite such advantages, concerns about potential memory corruption vulnerabilities constantly increase because of poor programming practices such as spatial violation (*i.e.*, pointer access to an illegitimate boundary), temporal violation (*i.e.*, pointer access to an invalid object), and unsafe type conversion (*i.e.*, `dynamic_cast`).

Lately, new (compilation-based) programming languages have been introduced to resolve the above downsides. One of the most popular emerging programming languages is Rust [13] with the notable feature of an ownership that controls the scope of every value at compilation. This ensures memory safety by enforcing that all references point to a valid memory region without relying on a garbage collector or a reference counter. Linux welcomes Rust, which officially supports it from the kernel version 6.1 [14]. Likewise, Go [15] and Nim [16] are gradually on the rise [17], [18]. Go offers both memory-safe and type-safe features, whereas Nim provides optional checks with a reference trace at compilation. Meanwhile, malware authors have been adopting emerging compiler toolchains for writing a malicious code [19], [20], [21]; *e.g.*, Agenda [22] with Rust, EKANS [23] with Go, and IceXloader [24] with Nim. There has been a 20 times increase of new malware written in Go over the past few years [20]. With the emerging compiler toolchains, the binaries compiled with them will become commonplace.

A wealth of study focus on identifying compiler toolchain provenance for GCC and clang [4], [5], [25], [26], [27], [28]. To identify a certain compiler toolchain (*i.e.*, compiler family and optimization level), a typical and straightforward approach utilizes a signature that represents the toolchain. A signature-based toochain identification method has been widely adopted as part of other tasks like decompilation [29], file type identification [30], or reversing [31], [32] due to its simplicity and effectiveness. However, the evident drawback of the signature-centered approach is to maintain appropriate signatures in a database according to a version update; the absence of a signature brings about detection failure. A plethora of recent advances mitigate such a signature-based downside by adopting machine learning techniques [4], [5], [6], [25], [26], [27], [33], [34], [35]. A deep neural network (DNN) gains attention including O-glassesX [26] with a convolutional neural network (CNN), NeuralCI [25] with a recurrent neural network (RNN), and BinProv [6] with the BERT architecture [36]. Lately, Du et al. [35] propose an ML-based LightGBM model, aiming to identify compiler optimization passes [37] However, none of the above can reveal the identification of emerging compiler toolchains. Adopting previous approaches is challenging because some emerging toolchains utilize the same backend of traditional compilers, which complicates the decision boundary of a model.

In this paper, we focus on the inference of both traditional and emerging compilation toolchains that have been rarely explored, including: Rust [13], Go [15], and Nim [16] as well as GCC [38], clang [39]. We thoroughly study both the features of an emerging toolchain itself and those of a (stripped) binary generated by the toolchain. One of our findings is that each toolchain inevitably inserts a considerable amount of subroutines (*i.e.*, binary functions) within an executable, which can be fingerprintable (as a birthmark). We leverage common functions for each toolchain across different versions into further toolchain identification.

To this end, we present ToolPhet, a BERT-based [36], [40], [41] end-to-end system for the inference of an emerging compiler toolchain. ToolPhet consists of four components that play a role ① to prepare a dataset (*e.g.*, disassembled instructions), ② to generate a generic model for assembly (*e.g.*, compiled with GCC, clang, Rust, Go, and Nim), ③ to create a specialized model for recognizing a compiler toolchain, and ④ to infer a toolchain with a target code snippet. We build two fine-tuned models for deducing compiler provenance: a toolchain classification model and a binary code similarity detection model. The classification model aims to classify a toolchain by directly learning the internal features of the toolchain. However, such a model may be misjudging in case of the same compiler backend (e.g., Nim adopts the GCC backend). In this case, we leverage a binary code similarity detection technique [42], [43] that infers the semantics of (compiler-specific) binary code to achieve our goal. To this end, we collect a list of birthmark subroutines per toolchain for further comparison. Namely, with the pre-defined list of our interest, ToolPhet can determine a toolchain based on the number of similar/dissimilar functions within a target binary.

To evaluate ToolPhet, we collect source codes written in Rust, Go, and Nim from GitHub [44] and Rosetta [45], followed by compiling them with a different version of each toolchain. We compare the ToolPhet's classification model with the five previous approaches including a signature-based one (*e.g.*, DIE [46]) and four machine-learning-based models (*e.g.*, RNN [47], LSTM [48], O-glassesX [26], and NeuralCI [25]). Our experiments demonstrate that a toolchain classification model outperforms (or is comparable to) prior approaches with F1s of 0.979, 0.996, 0.996, 0.968, and 0.956 for Rust, Go, Nim, GCC, and clang, respectively.

Our main contributions are summarized as follows:

- We propose ToolPhet, an efficient end-to-end BERT-based system for the provenance inference of both traditional and emerging compiler toolchains (*e.g.*, Rust, Go, Nim, GCC, clang).
- We thoroughly study not only the properties of each emerging toolchain but also those of executable binaries built with those compilers, which can help further toolchain prediction.
- We design and implement ToolPhet with two downstream tasks (for achieving the same goal): a toolchain classification model and a code similarity detection model (by leveraging the inference of code semantics).
- We evaluate ToolPhet with varying approaches including a signature-based tool and four machine-learning-based models, which achieves comparable or better performance.

II. RELATED WORKS

This section summarizes prior work with three approaches pertaining to provenance inference on a binary: compiler toolchain, programming language, and compiler optimization pass. Note that this work focuses on the toolchain inference for emerging compilers.

A. CONVENTIONAL COMPILER TOOLCHAIN INFERENCE APPROACHES

Before Rosenblum et al. [5] pioneer the (separate) problem of identifying a toolchain provenance from a stripped binary, obtaining compiler information has been part of other tasks like decompilation [29] and file type identification [30]. One of the straightforward means is a signature-based approach because of its simplicity in a cost-effective manner. Today, a majority of reverse engineering tools such as IDA Pro [31] and Ghidra [32], can infer toolchain information with known signatures. However, the downside is that the maintenance of a signature database requires continuous efforts and time where a missing signature is undetectable. Thus, we decide to adopt an ML-based compiler toolchain inference approach.

B. ML-BASED COMPILER TOOLCHAIN INFERENCE APPROACHES

Early works suggest two machine-learning-based models with a conditional random field (CRF) [5] and a support vector machine (SVM) technique [4], which identifies the characteristics of a binary code pertaining to a specific compiler toolchain. On the other hand, recent advances have begun to utilize a deep neural network. O-glassesX [26] proposes a model based on convolutional layers [49] (for capturing every bit in the whole instruction) and Attention [50] at the last layer (for identifying each instruction contribution on a toolchain). The model implementation comes with the following two options: with and without disassembly. The former case disassembles a binary code, converting each instruction into a fixed-length instruction (filled with constant bits). The latter works directly on an instruction sequence, attempting to recover the origin of a compiler toolchain. Meanwhile, NeuralCI [25] introduces a model with an RNN to reveal compilation information in detail (*e.g.*, compiler type, optimization level, compiler version of an individual function). Benoit et al. [27] harness a control flow graph for building a neural network model for predicting a toolchain at the binary level. Lately, BinProv [6] is probably closest to ours in terms of utilizing a BERT-based model for classifying a compiler (*e.g.*, GCC, clang) with an optimization level. To the best of our knowledge, we first deal with emerging compiler toolchains including Rust, Go, and Nim, in the field of ML-based compiler provenance inference.

C. PROGRAMMING LANGUAGE INFERENCE

Adhikari et al. [51] focuses on detecting a programming language itself with string metadata from binaries such as Swift, Rust, C/C++, Go, and Fortran, which differs from our goal to identify a compiler toolchain.

D. COMPILER OPTIMIZATION PASS INFERENCE

Lately, Yufei et al. [35] utilize the LightGBM [37] to discern compiler optimization passes applied to binary files. By leveraging the compiler passes information applied to each function during compilation, they construct varying inference models specific to each individual compiler pass for further identification. With the inferred pass information, (un)known vulnerabilities and the occurrence of code reuse gadgets are revealed. Note that this work differs from ours as focusing on a compiler optimization pass inference.

III. BACKGROUND

This section describes the preliminaries to build our end-to-end system for identifying a compiler toolchain.

A. EXECUTABLE CODE GENERATION

A human-written source code must be converted into a series of machine instructions that a processor can eventually fetch and execute. This executable form can be achieved through different conversion methods: ① a compiler, ② an interpreter, or ③ a virtual machine (VM). The first type is a compilation process that takes one or more source code and converts each into an object file, followed by producing a final executable (*i.e.*, native code) that can be run under a certain architecture (*e.g.*, C [52], C++ [53], Objective-C [54], Rust [13], Go [15], Nim [16]). A compiled language (*e.g.*, C [52], C++ [53], Objective-C [54], Rust [13], Go [15], Nim [16]) involves intricate transformation and optimization facilitated by a compiler toolchain (*e.g.*, compiler, linker, assembler). The second type is an interpreter-based language (*e.g.*, Python [55], Ruby [56], R [57], Perl [58], Lua [59]), which directly parses source code line by line from a high-level program. Note that an interpreter requires a source code for its execution at all times as machine code is not available. While a compiler-generated program is fast, machine-specific, and inflexible, an interpreter-based program is slow, machine-agnostic, and flexible.

A VM-based language (*e.g.*, Java [60], C# [61], JavaScript [62]) attempts to resolve such a trade-off by producing a bytecode that can be executed under a virtual machine (*e.g.*, JVM [63] for Java, V8 engine [64] for JavaScript) as an intermediate interpreter for performance improvement. This paper focuses on a compilation-based toolchain because an interpreter-based toolchain with either a source syntax or VM bytecode is relatively easy.

B. BINARY CODE SIMILARITY DETECTION

A task of detecting binary code similarity is to determine whether two code snippets are similar when the original source is unavailable. BinShot [9] is a binary similarity detection framework that utilizes both the BERT (short for Bidirectional Encoder Representations from Transformers) [36] architecture and the Siamese network structure. BERT is initially designed for interpreting and deriving meaning from human languages (*e.g.*, text, voice) in the field of NLP, assisting varying common language tasks including sentiment analysis, question-and-answer tasks, language translation, etc. BERT takes two training phases; one for building a generic model, and another for building a specific model tailored to a downstream task. The design details of the BERT model is out of this paper's scope. A Siamese neural network consists of a twin network that utilizes the same weights in tandem on two input vectors to produce an output with a distance function. For example, a set of similar pairs would be encoded as adjacent embeddings whereas that of dissimilar pairs as distant embeddings. The idea of BinShot [9] lies in learning a distance itself with a *weighted* distance vector (distance function) and a binary cross entropy function (loss function) for better binary similarity detection. Note that we take advantage of BinShot to confirm if a function of our interest (*e.g.*, particular function that can represent a compiler toolchain) is within a binary in question.

C. WELL-BALANCED INSTRUCTION NORMALIZATION

Unless feeding a series of byte codes itself into training a model, one must disassemble them in a human-readable machine instruction. Previous work [65], [66], [67] illustrate that these disassembled instructions can be treated as a vocabulary, resembling a language model. However, a naïve approach such as a one-hot encoding [68] or even word embedding strategy (*e.g.*, word2vec [69]) suffers from handling the total number of vocabularies, causing an out-of-vocabulary (OOV) and a sparse vocabulary problem because of a tremendous number of possible tokens (*i.e.*, mnemonic, operand, or both). This hinders the update of each token's embedding during a backpropagation algorithm. In this regard, it is common to normalize an instruction to reduce the number of tokens (*e.g.*, immediate value → `imm`, targets in a callsite → `foo`). Our work adopts a well-balanced instruction normalization technique [70], which attempts to strike a balance between the expressiveness of binary code that preserves the original code semantics and the problems from vocabularies like OOV and sparsity.

D. FUZZY HASH

A hash function is a mathematical algorithm that maps a variable-length value in a large domain to a fixed-length one within a relatively small domain, while holding the following two properties: ① one-wayness (*i.e.*, reverse mapping is not viable), and ② collision resistance (*i.e.*, finding two arbitrary inputs that map to a certain value is computationally infeasible). Unlike a cryptographic hashing algorithm that meets the above properties for integrity, a fuzzy hash [71] allows one to be aware of similarity by computing piecewise hashes of each part. For instance, an input file can be split into multiple parts to produce a hash of the whole file, thereby being capable of searching for similar messages. A fuzzy hash is often used in malware classification and clustering. In this work, we utilize the fuzzy hash to check how the body of a function (*i.e.*, assembly code) differs from another, in which the two functions hold the identical function symbol names (from a debugging table) across different compiler toolchain versions.

IV. PROBLEM AND APPROACH

This section concisely describes the problem of deducing compilation toolchain provenance, followed by its goal, scope, and our approach.

A. PROBLEM DOMAIN

The toolchain provenance problem is to identify a compiler toolchain from a stripped executable binary. A wealth of previous approaches [4], [5], [6], [25], [26], [27], [33], [34], [50] attempt to infer toolchains for stripped binaries, primarily focusing on identifying GCC and clang toolchains, with an emphasis on languages compiled from sources written in C and C++ programming languages. However, the problem for emerging compiler toolchains has been yet explored in the literature. One may argue that deducing a toolchain would be feasible from specific section names or other prominent features. However, such an approach is not reliable because such information could be simply updated (*i.e.*, renaming a section name). For example, an executable binary with Go [15] creates a section named `.note.go.buildid`. While being easily identifiable, modifying the section name is trivial (*e.g.*, `objcopy`'s `-rename-section` option). Hence, we narrow down the problem into the inference of a compiler toolchain when a sequence of bytes (*i.e.*, machine instructions) are given.

B. GOAL AND SCOPE

We aim to build an end-to-end system that can infer the toolchain provenance (*i.e.*, output) from a given binary compiled with either traditional or emerging compiler toolchain (*i.e.*, input). In this paper, we concentrate on three emerging compilation-based programming languages including Rust [13], Go [15] and Nim [16], as well as GCC

and clang that have been covered by previous approaches [5], [6], [25], [26], [27], [28]. While the emerging toolchains offer a slightly deviating options for optimization levels,[1] we conduct our experiments on optimized binaries based on the assumption that most COTS executable binaries have been built with a release mode by default [72]. Our system currently supports the Intel's `x64` architecture, but it can be expanded to support other architectures in the future. It is worth noting that we exclude obfuscation techniques as they fall beyond the scope of our current work.

C. CHALLENGES

Due to the nature of the absence of high-level information at a stripped binary, recognizing the toolchain from a byte stream or a low-level assembly language is challenging.

D. RULE-BASED APPROACHES

A straightforward approach would be deterministically inferring a compiler toolchain with a pre-defined rule. For instance, PEiD [73] can detect a packer, cryptor, or compiler of a given PE [74] binary based on a database of diverse signatures (*e.g.*, scanning a preset mask on bytes). Similarly, Detect It Easy (DIE) [46] extends a signature-based capability by defining a programmable user-defined rule, which is another signature-based tool to determine the properties of a file such as a type (*e.g.*, PE [74], ELF [75], MACH [76]), packer (if packed), compiler, linker, base address, and its entry point. However, rule-based approaches necessitate continuous maintenance of new signatures or algorithms for compiler toolchain identification. Another downside is a false negative case, in which a signature that represents a feature is missing (although a false positive rate would be negligible).

E. OUR APPROACH

We utilize BERT, which is a state-of-the-art architecture in the field of natural language processing (NLP), to automatically learn hidden features of the underlying assembly language and address the problem of compiler toolchain recognition. Our approach leverages the capabilities of the BERT model [36], [40], [41], which accumulates substantial code semantics during its pre-training phase. However, in a binary code snippet, a uniform instruction may not consistently convey the same meaning, as its behavior varies based on the context of the instruction. The pre-training phase of BERT customized for binary characteristics assists in understanding relationships between instructions within a function. Furthermore, it facilitates the creation of models for various purposes, including classification and similarity models in our specific context. The strengths of our approach are twofold: ① it demonstrates enhanced robustness in identifying new, unseen versions of a toolchain, and ② it exhibits lower false-negative rates compared to rule-based methods. This is attributed to the model's ability to unveil hidden features, such as code semantics. In addition, we build two fine-tuned models - a toolchain provenance classification model and a binary similarity detection model - because the former may not suffice when the emerging toolchain adopts the existing compiler backend like Nim.

[1] Rust provides similar optimization levels (*i.e.*, O0-O3) to GCC or clang where 0 for a debugging mode and 3 for a release mode. Go comes with an optimization by default, providing an option to merely switch it on and off. Nim provides optimization levels according to a size, speed, debug or release mode.

V. EMERGING COMPILER TOOLCHAINS

This section describes a brief history of emerging (compiled) languages that are on the rise. Note that we survey the main features of an emerging compiler toolchain (*i.e.*, Rust [13], Go [15], Nim [16]), which can be the birthmark of a binary for assisting our toolchain recognition task.

A. RISE OF EMERGING COMPILED LANGUAGES

Although it is difficult to measure accurately the popularity (*i.e.*, widely used) of a programming language, statistics like the TIOBE index [18] shows that the C [52], C++ [53], and Java [60] programming languages have been dominant over the last 20 years. Focusing on a compiled language (without requiring an interpreter), C and C++ remain the most widely adopted (general-purpose) programming languages because of speed, portability, clean support for a target CPU, and varying standard libraries. With such advantages, core system components (*e.g.*, operating system, device driver, protocol stack, compiler, interpreter) have been implemented in C/C++. The popularity of compilation toolchains has been aligned with the creation of executable binaries in these languages, encompassing GCC [38], clang [39], oneAPI DPC++ [77] (previously known as Intel ICC[2]), and Visual Studio [78] compilers. While C/C++ excel in efficiency and flexibility, offering low-level features for memory access, the persistent memory safety issue [79] remains a significant concern. In this paper, we choose three emerging compiler toolchains to address such issues: Rust [13], Go [15], and Nim [16].

B. RUST FEATURES

Since the first stable release around 2015, Rust [13] becomes one of the most popular programming languages, emphasizing a safe, reliable, efficient, and manageable software build. The most unique feature of Rust is an *ownership* that allows a single owner to hold every value, which controls the lifetime (scope) of a value. This concept facilitates the seamless development of both memory-safe (*e.g.*, memory allocation and deallocation without a garbage collector, no support for a null/dangling pointer) and thread-safe program (*e.g.*, transiting thread ownership from another can prevent data races and deadlocks for concurrency). Moreover, a strictly typed language supports a type-safe feature by restricting a function parameter to an annotated type. Notably, Linux began to support Rust in its kernel from version 6.1 [14].

[2] Intel C++ Compiler Classic (ICC) has been deprecated and will be replaced by a oneAPI release as of 2023.

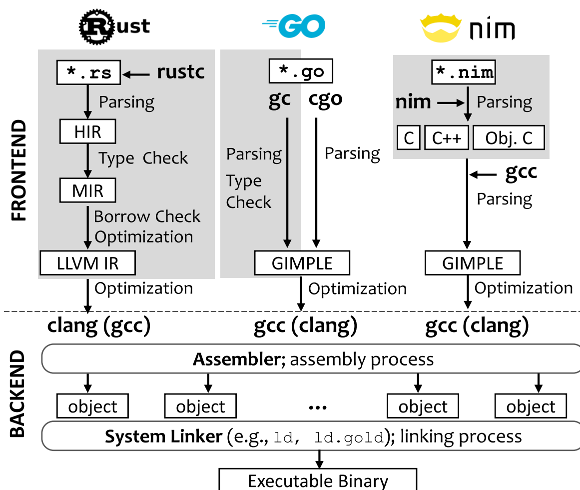

FIGURE 1. Emerging compiler toolchains leverage the existing backend (*e.g.*, GCC, clang) into building a final executable. The gray boxes denote each toolchain's role. Rust introduces new intermediate representations (*i.e.*, HIR, MIR) for type/borrow checking, generating LLVM IR to be fed into the clang backend. The Go compiler has a similar checking routine to create GIMPLE (GCC IR) for the GCC backend. Note that GCC can directly take go source files with `cgo` for compilation. Nim translates a nim source into C/C++/Objective C, which the GCC frontend takes it as an input. Note that Rust utilizes clang by default, whereas Go and Nim utilize GCC.

C. RUST COMPILER

In essence, the Rust compiler (*rustc*) aims a reliable and quick compilation while consuming less memory like existing GCC/clang toolchains. The key distinction lies in various checking routines being part of the frontend building process, including type confirmation and memory access. To this end, rustc introduces several layers (Figure 1) to generate a high-level intermediate representation (HIR) for type checking, followed by creating a mid-level representation (MIR) for borrow checking. Next, rustc converts MIR into LLVM IR [39] for further optimizations, leveraging the full LLVM backend to produce a series of machine instructions. While the Rust compiler defaults to using LLVM as its backend, it starts supporting the `GCC` backend from mid-2021. There are two ways to build a Rust program with either ① `rustc` or ② `Cargo`. The former directly invokes the Rust compiler, and a user should maintain dependencies (*e.g.*, library) while the latter supports an automatic build system with handy package management. For example, `Cargo` manages necessary library dependencies, making it the preferred choice for most Rust developers for easy package building.

D. GO FEATURES

Since its first release in 2012, the Go programming language [15] has been designed for fast, efficient, and practical programming to support large-scale development. First, Go generates a *static* binary by including all runtime code (*e.g.*, memory allocation, garbage collection), which does not require any dependency at runtime. By design, a single-executable-contains-all strategy removes the hassle of maintaining dependencies in a central repository like NPM [80] for JavaScript or `crates.io` [81] for Rust. Second, Go internally maintains a garbage collector for recycling memory on behalf of an application, which is memory safe.[3] Third, part of the Go implementation entails an assembly for performance, including `math`, `reflect`, `syscall`, `runtime`, and `crypto` packages. Fourth, Go attempts to allocate data (*e.g.*, object) on the stack if possible, which a GC process deallocates it when a reference is no longer available. Finally, like Rust, Go adheres to a statically-typed language, and supports type-safety, memory-safety, and concurrency (with `goroutines` and `channels`). These features allow for discovering quite a few common functions across binaries that are built with Go.

E. GO COMPILER

One can build a Go executable in two ways: `gc` and `GCC go` (Figure 1). Go utilizes the `gc` compiler by default. Another option is using `GCC go` when combining Go with existing C code is needed. The latter case allows one to reuse (well-written) previous C code (*e.g.*, shared libraries) with a dynamic linking mechanism. `GCC go` requires an additional installation to support both compilation and linking flags (*e.g.*, `CFLAGS`, `LDFLAGS`). Notably, there were major architectural changes since version 1.5, with the Go compiler and runtime fully transitioning to being written in Go itself (versions prior to 1.5 were written in C [82]). In this regard, we select all versions later than Go v1.5 for our experiment. Go utilizes the GCC backend by default, however, Gollvm [83] supports an LLVM-based compiler for the Go language.

F. NIM FEATURES

Nim [16] is a relatively young programming language (*i.e.*, the first release in 2008), which is under active development [84] until today as an open source. One of the advantageous and core features in Nim is extensive metaprogramming support[4] (*e.g.*, generics, templates, macros), which enables one to analyze and manipulate source code with ease. For example, it can eliminate boilerplate code (*i.e.*, repetitive code all around a program). Nim offers a friendly interface that can integrate with other languages (*e.g.*, C, C++, Objective C, JavaScript) via its foreign function interface (FFI) like Java Native Interface (JNI) [85] in Java or Platform Invoke (P/Invoke) [86] in C#. Besides, Nim offers optional checks at compilation for memory safety (*e.g.*, reference trace) [87] by disabling a pointer arithmetic with a garbage collector.[5]

[3] Note that the memory safety level in Go is slightly different from that in Rust because Go thwarts an unsafe status via panicking (*e.g.*, `panic` function) and terminate a program when things go wrong, like an out-of-bound access.

[4] Metaprogramming allows for generating a domain-specific language (DSL).

[5] Nim supports a different type of garbage collectors of one's choice, meaning that a developer may lift it for performance (*e.g.*, resource-hungry devices).

G. NIM COMPILER

The Nim compiler directly converts Nim code into several languages, including C, C++, Objective-C, and JavaScript (Figure 1). This implies that Nim harnesses both frontend and backend of the existing compiler and linker once the source code conversion is complete. Note that Nim inserts code pertaining to memory management (*e.g.*, garbage collector) at this conversion (*e.g.*, C code generation) phase. This property allows us to extract common binary functions in the binaries that are built with Nim. Furthermore, Nim supports cross-compilation, which generates an executable binary under multiple platforms (*e.g.*, MS Windows, *Nix, Mac OS). The executable contains a native code that is free from a virtual machine dependency, ensuring little overhead (*i.e.*, efficiency) and handy redistribution (*i.e.*, portability). Nim utilizes the GCC backend by default, however, nlvm [88] supports an LLVM-based compiler for the Nim language.

VI. EXECUTABLE WITH AN EMERGING TOOLCHAIN

This section explores the properties of a binary that has been built with an emerging compilation toolchain, which assists in building an appropriate model with BERT.

A. FUNCTION INVESTIGATION

ToolPhet utilizes the BERT architecture with two options for a fine-tuning process: a classification model and a binary similarity model. We thoroughly study on the functions in common by default, which assists in inferring code semantics for the similarity model.

1) INSPECTING DEFAULT FUNCTIONS

As alluded in section V, each emerging compiler toolchain inevitably generates a bloated binary by adding quite a few (default) subroutines, resulting in a noticeable increase in code size. We write a simple code per language with an empty function (*e.g.*, `main()`), building an executable. This allows us to identify default functions that have been inserted by a different toolchain. As a baseline, we build four executable binaries with GCC 7.5.0, 9.4.0, and clang 6.0.0, 10.0.0. We observe that the functions in those binaries are mostly inserted by a system linker (*e.g.*, `ld`) such as the C runtime (CRT) routines (*e.g.*, `_start`, `(de)register_tm_clones`, `frame_dummy`, `__libc_csu_init`, `fini`). Table 1 demonstrates the comparison of the baseline with an executable binary built with varying emerging toolchains. We chose a handful of different versions considering a release date, version interval, and feature update, accordingly. Our finding shows a distinct code increase (in terms of the number of functions) due to supporting the feature of an emerging toolchain. Note that 1.5 or later versions of Rust are purposefully selected due to a major update (*i.e.*, the whole compiler has been written in Rust itself while in C for prior versions).

TABLE 1. Comparison of the number of default (binary) functions between an emerging compiler toolchain and an existing one. As a baseline, we build an executable with an empty `main()` function. We observe that the code size distinctively increases by an order of magnitude (10-100 times), indicating the emerging toolchains tend to add varying feature codes.

Toolchain	Compiler Version	Release Date	Number of Functions	Code Size (Bytes)
Rust	1.52	May.06,2021	416	0x2f423
	1.56	Oct.21,2021	462	0x33cd3
	1.58	Jan.13,2022	461	0x33c33
	1.62	Jun.30,2022	529	0x35643
Go	1.10	Feb.16,2018	955	0x4b1af
	1.12	Feb.25,2019	981	0x4e990
	1.15	Aug.11,2020	1,021	0x5ccf0
	1.18	Mar.15,2022	999	0x54270
Nim	1.0.10	Oct.27,2020	129	0xd7a5
	1.4.8	May.25,2021	151	0x13f25
	1.6.10	Nov.23,2022	148	0xe725
GCC	7.5.0	Nov.04,2019	8	0x195
	9.4.0	Jun.01,2021	8	0x175
clang	6.0.0	Mar.08,2018	9	0x172
	10.0.0	Mar.24,2020	9	0x175

2) RECOGNIZING COMMON FUNCTIONS

Identifying *common* (binary) functions across the same version of a compiler toolchain is of significance because they play a crucial role to offer a clue of its provenance. We hypothesize that it is possible to deduce compiler toolchain provenance if a set of common functions were observed in a binary at all times. Our hypothesis lies in the observation that a set of common (and special-purpose) routines would be inserted by a compiler toolchain. However, this is not always true because functions that have an identical (symbol) name may vary depending on the version according to our thorough analysis, which we revisit in section VII-B. To this end, we collect our own dataset (section IX-A), generating hundreds of executables. Then, we record actual function names with mapping information from a linker (*i.e.*, linker map) that provides an option (*e.g.*, `-M` in `ld`) to output the mapping of a code section (*i.e.*, virtual address of an address-taken function). For example, in Rust, we can obtain full mappings with

```
rustc -g -o out -C link-args=
"-Wl,-Map=output.txt" src.rs
```

when consolidating all internal object files.

3) EXPERIMENTAL RESULTS

We discover that (at least) hundreds of functions are present by default when a binary has been built with a certain version of Go, Nim, and Rust, which can be the birthmark of a compiler toolchain. Table 2 summarizes the number of common functions per each toolchain and version. For example, 999 functions are observed in common with Go 1.18 by their function symbol names, and 242 common functions are present across all different versions.

TABLE 2. The number of common functions across different versions per emerging toolchain. We filter out functions (with our own dataset in section IX-A that are appeared from all versions by function symbol name. Then, we examine the similarity of those functions (in terms of code semantics) with fuzzy hash, revealing that 60, 54, and 57 common functions that can be fingerprintable for our purpose (Section VII-B).

Toolchain	Version	Number of Common Functions
Rust	1.52	340
	1.56	386
	1.58	385
	1.62	438
	All versions	251 (60)
Go	1.10	955
	1.12	981
	1.15	1,021
	1.18	999
	All versions	242 (54)
Nim	1.0.10	129
	1.4.8	145
	1.6.10	142
	All versions	53 (57)

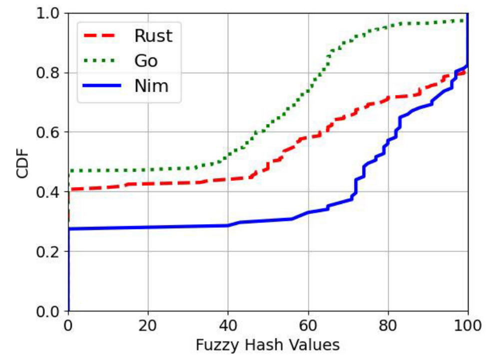

FIGURE 2. CDF of fuzzy hash values with common functions across different versions of each emerging compiler toolchain. We generate all function pairs by grouping identical functions according to a function name, computing the fuzzy hash value of each pair. Interestingly, quite a few pairs turn out to have a different function body (*e.g.*, version updates). We choose similar function pairs (*e.g.*, Hash value > 70) to build a binary similarity model with the dataset.

B. EXECUTABLE BINARY FEATURES

This section briefly covers the features of an executable that is built by each emerging toolchain.

1) RUST BINARY

As in Table 2, a set of unique functions are discovered for each different version. Our observation shows that common functions from the binaries of our choice (Table 3) include runtime functions (*e.g.*, `std::panicking` package), name mangling functions (*e.g.*, `rustc_demangle` package), core package functions (*e.g.*, `core::fmt`, `core::ptr`, `core::slice`, `core::str`), and system functions (*e.g.*, `std::sys`, `std::sys_common`, `std::thread`). In Rust binaries, reliance on packages (and their functions) imported through source code is commonplace. Note that a Rust function name adheres to its designated mangling rule [89]. A Rust binary integrates varying built-in functions like `core::panicking::panic_boundscheck` that is dedicated to examining the size of dynamically allocated objects to prevent memory errors.

2) GO BINARY

Akin to Rust, producing a static Go binary comes with a number of (built-in) language-specific sub-routines, which unavoidably increases its size. The common functions in Table 2 predominately belong to the following packages (for garbage collection, scheduling, concurrency) including `fmt`, `os`, `reflect`, `runtime`, `strconv`, `syscall`, `type`, `unicode`, etc. For instance, Go utilizes `goroutines` for concurrency programming, which is invoked by the `runtime.schedinit` function. Additionally, certain package implementations are composed in (architecture-dependent) assembly for enhanced performance, serving as another distinctive characteristic of a Go binary when the package is utilized. Interestingly, a Go build specifies an entry point function that relies on a system architecture (*e.g.*, `rt0_linux_amd64` in Linux).

3) NIM BINARY

Unlike Rust and Go, the Nim compiler converts a nim source to the existing languages (*e.g.*, C, C++, Objective C), followed by taking full advantage of the frontend (*e.g.*, IR generation, optimization) and backend (*e.g.*, instruction selection, assembly) of either GCC or clang (Figure 1). Hence, we observe that a number of subroutines are embedded all the time for supporting Nim features in a converted source code; *e.g.*, functions for garbage collection: (`initGC`, `nimGC_setStackBottom`, `nimGCvisit`), functions for checking an overflow (`raiseOverflow`), functions for maintaining a data type and a system module (`appendString`, `addChar`, `addInt`, `alloc`, `dealloc`). Note that a converted C source can be found at the location of `$HOME$/.cache/nim/[binary_name _r|d]` (r for release, and d for debugging). Another unique property of a Nim binary is that all user-defined functions are invoked within the `NimMainModule` function (by `NimMainInner`).

VII. TOOLPHET DESIGN

A. OVERVIEW

Figure 3 depicts the overview of ToolPhet that consists of four phases. As a preprocessing process, we identify function boundaries from an executable binary, disassemble every function with IDA Pro, and normalize it for being properly fed into a neural network (Section VII-B). At this phase, we prepare all required dataset for further model generation, including tokenized disassembly, task-dependent dataset with a label. Next, ToolPhet adopts the BERT architecture [36], [40], [41] that requires two training steps: pre-training for generating a generic model (Section VII-C), and fine-tuning for generating a specific model tailored for a downstream task (Section VII-D). To tackle a compiler toolchain provenance

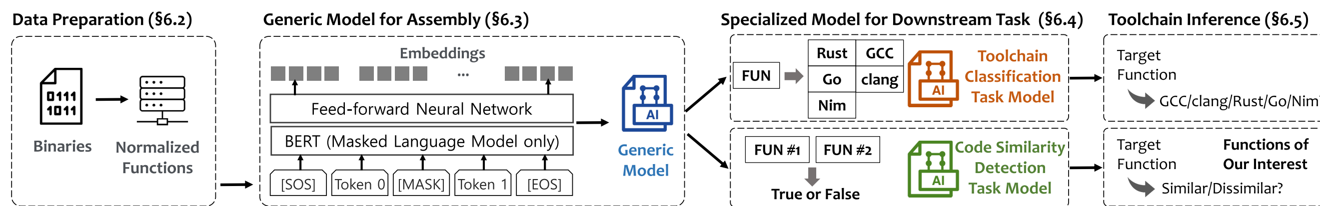

FIGURE 3. Overall ToolPhet workflow that consists of four components: preparing a dataset (Section VII-B), creating a generic model for assembly (Section VII-C; known as pre-training in BERT), generating a special model for compiler provenance (Section VII-D; known as fine-tuning), and toolchain inference (Section VII-E). Note that ToolPhet defines two separate downstream tasks for the same goal, generating a classification model and a similarity detection model. The former directly infers a toolchain, while the latter determines if a group of functions are similar to the pre-defined functions that can represent each toolchain (Section VI-B).

problem, we define two downstream tasks: ① a classification task that directly learns the properties of a function that belongs to an emerging toolchain, and ② a binary similarity detection task that determines a toolchain by confirming the presence of a set of unique functions in a binary.

B. DATA PREPARATION

1) INSTRUCTION NORMALIZATION

We leverage IDA Pro [31], one of the most popular reverse engineering tools, to recognize the boundary of a function, followed by disassembling the function. Then, we utilize a well-balanced instruction normalization [70] for better binary code representations, ensuring the preservation of the original code semantics while mitigating out-of-vocabulary (OOV) concerns. This normalization technique takes a single instruction (*i.e.*, opcode and zero or more operands) as a token.

2) FINGERPRINTABLE FUNCTIONS WITH FUZZY HASH

Recall that ToolPhet defines two downstream tasks; one of them is a binary similarity detection task (during fine-tuning). This task involves assigning labels to pairs of functions to denote their similarity or dissimilarity. A naïve approach would be generating a function pair where two function symbol names are identical, however, we discover that quite a few functions are not consistent upon a version update. In other words, the same function (by name) could have a different body (*i.e.*, instructions) across different versions of a toolchain. We harness a fuzzy hash algorithm to check how two functions match each other. Figure 2 illustrates a cumulative distribution function (CDF) of fuzzy hash values from each compiler toolchain, revealing that a significant number of function name pairs (initially assumed to be identical) are indeed different. For example, approximately 40%, 50%, and 30% function pairs of Rust, Go, and Nim are considerably different (*i.e.*, fuzzy hash value $= 0$), respectively. This is probably because of adding, deleting or altering the content or structure of a function while a version update. Hence, we choose a fraction of the whole function pairs with a threshold (T) or above (*i.e.*, similar function body) for generating a binary similarity dataset. To this end, we extract a list of *fingerprintable functions in common* whose fuzzy hash values are above a certain threshold (*e.g.*, $T = 70$) (*i.e.*, changes between versions are not significant). We finally collected 60, 54, and 57 common functions that meet the threshold for Rust, Go, and Nim compilers, respectively. Note that we utilize these functions of our interest when creating a model for binary similarity detection.

3) EXCLUDED FUNCTIONS

As an emerging toolchain entails solely the frontend while adopting the backend of existing toolchain, we exclude all functions inserted by a linker (*e.g.*, C Runtime functions).

C. GENERIC EMBEDDING MODEL FOR ASSEMBLY

ToolPhet follows the BERT scheme that creates a generic model via pre-training. Conceptually, similar to a natural language processing domain, we view an instruction (*i.e.*, token) as a word, and a function as a sentence. This step learns the relationships of different instructions (*i.e.*, assembly language) within a function. The original BERT incorporates both the Masked Language Model (MLM), which masks a fixed portion of the input (*e.g.*, 15%) and Next Sentence Prediction (NSP). However, we rule out NSP because the proximity of a function is not determined by its location but the relationship of a call invocation (*i.e.*, caller and callee with `call` and `ret` instructions). It is noteworthy to mention that we retrained a generic model with our own dataset because the state-of-the-art model from BinShot [9] utilizes binaries that are compiled with GCC or clang.

D. SPECIALIZED MODELS FOR DOWNSTREAM TASK

The BERT scheme allows one to fine-tune a generic assembly embedding for inferring a compiler provenance task, which generates specialized models. Note that a separate dataset with a label is required (*i.e.*, supervised learning) for fine-tuning. We adopt two approaches as a fine-tuning task: toolchain classification task and binary code similarity detection task.

1) TOOLCHAIN CLASSIFICATION (TC) MODEL

The toolchain classification model directly learns the internal features (*i.e.*, relationships between tokens) of each emerging language on top of the generic assembly embedding model (Section VII-C). The model emits the inference of five

toolchain categories when a function is given. For computing optimal network parameters (θ) at fine-tuning layers, we use cross-entropy as a loss function as follows:

$$\theta = \arg\min_{\theta} \sum_{c \in C} p(c|y) log(p(c|\hat{y})) \tag{1}$$

2) CODE SIMILARITY (CS) DETECTION MODEL

The binary code similarity detection model aims to predict if a given code snippet (*i.e.*, assembly function) is similar to a compared one. We set a code granularity to be a (binary) function, feeding it to the model as an input. For further comparison, we pre-define common functions of our interest from each compilation toolchain, which can be the birthmark of the toolchain. As described in Section VII-B, we filtered out a fingerprintable function with the fuzzy hash for training a similarity detection model. The weights are adjusted based on the generic model (Section VII-C) so that a binary classifier can learn a weighted distance vector with a binary cross entropy loss function as introduced by BinShot [9]. Note that the weighted distance differs from a scalar distance like a cosine similarity between two vectors. The distance vector between two functions (*i.e.*, X and Y) is learned during training with the following equation where D represents the Euclidean distance embedding in the n dimensional space (Equation 2). Likewise, when two functions (*i.e.*, X_i and X_j) are given, the distance function, $F(X_i, X_j)$, can be obtained from a fully connected layer where X_i and X_j represent the two arbitrary functions from the pretrained BERT model (Equation 3).

$$D(X, Y) = \{e_1, \ldots, e_n\}, e_i = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2}$$

$$F(X_i, X_j) = \sigma(D(X_i, X_j) \cdot W + b) \tag{3}$$

Then, we can determine the similarity of the two functions with a threshold of C (*e.g.*, 0.5); similar if $F(X_i, X_j) \geq C$ or dissimilar otherwise.

E. INFERENCE OF TOOLCHAIN PROVENANCE

The process to determine the compilation toolchain with a number of functions in a given binary is slightly different depending on a specialized model.

1) INFERENCE WITH THE TC MODEL

The toolchain classification model takes a set of (normalized) functions in a target binary, and then produces the inference of compilation tools for each function (*e.g.*, Rust, Go, Nim, GCC, clang). For instance, Deadbolt [90] has 2,383 functions, resulting in `Rust: 2`, `Go: 2365`, `Nim: 0`, `GCC: 8`, and `clang: 8`. In this case, we compute the confidence of the Deadbolt's toolchain being Go with 0.992.

2) INFERENCE WITH THE CS MODEL

In the case of the code similarity detection model, we randomly select $K\%$ of entire functions in a binary, comparing them with (pre-selected) common functions that represent a toolchain. This is because ① common functions are appeared in a location-agnostic fashion, and ② unbalanced similar/dissimilar pairs may distort a prediction; *i.e.*, dissimilar function pairs would increase as with a number of user-defined functions, which results in growing the number of false positives. Furthermore, the performance of ToolPhet would be degraded with the number of comparisons; *e.g.*, the pairs of thousands of functions with hundreds of common functions could require a million times of comparisons. With a threshold (C) of the comparison results that are true (similar pair), we infer that a binary falls into the category of a certain toolchain.

VIII. IMPLEMENTATION

We develop ToolPhet with PyTorch 1.10.0 [91], one of the most popular frameworks for machine learning.

A. GLOBAL HYPERPARAMETERS

ToolPhet has three global hyperparameters, all of which are associated with a code similarity detection model: ① we elect a similar function pair whose body is larger than a fuzzy hash value of T for a training dataset, ② a small portion (K) of functions are chosen for testing (inference), and ③ we determine that a binary has been compiled with a certain toolchain when similar pairs are above C We set up $T = 70$, $K = 0.1$, and $C = 0.5$ for our experiment, which can be adjustable.

B. HYPERPARAMETERS FOR BERT

ToolPhet adopts the original strategy of BERT MLM by masking around 15% of the input tokens at random. We apply the following BERT hyperparameters to ToolPhet: 256 dimensions for instruction embeddings, 128 hidden layers, eight attention layers, eight heads, and a maximum input size of 256. Additionally, we utilize the Adam optimizer with a learning rate of 0.0005, and a dropout rate of 0.1.

IX. EVALUATION

This section evaluates ToolPhet with one deterministic approach (*e.g.*, DIE [46] tool) and five machine-learning-based models for comparison, including NeuralCI [25], O-glassesX [26], SAFE [92], LSTM [48], and RNN [47]. Note that we utilize SAFE [92] for function embeddings because it does not aim to directly perform any task.

Environmental Setup: We utilize the server for model training and evaluation on top of Ubuntu 20.04 with Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz (10 cores), 128 GB RAM, and Quadro RTX 8000 GPU card. We install CUDA that is compatible with PyTorch 1.10.0.

Evaluation Metrics: Let the number of true positives, false positives, true negatives, and false negatives be TP, FP, TN, and FN, respectively. Then, the metrics of precision (P), recall (R), F1 score (F1), and accuracy (A) can be computed

TABLE 3. We collect varying sources for GCC, clang, Rust, Go, and Nim, building 1,683 executables in total. We adopt a high optimization level or a release mode by default that can be seen in the wild. Note that we leave debugging information available for ground truth.

Toolchain	Version	# of Binaries	Optimization level
Rust	1.52	122	O3
	1.56	122	O3
	1.58	122	O3
	1.62	122	O3
	Total	488	(High optimization)
Go	1.10	100	Default
	1.12	100	Default
	1.15	100	Default
	1.18	100	Default
	Total	400	(Release mode)
Nim	1.0.10	101	Default
	1.4.8	101	Default
	1.6.10	101	Default
	Total	303	(Release mode)
GCC	7.5.0	125	O2
	9.4.0	121	O2
	Total	246	(High optimization)
clang	6.0.0	125	O2
	10.0.0	121	O2
	Total	246	(High optimization)

as following:

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ F1 = \frac{2PR}{P + R} \quad (4)$$

$$A = \frac{TP + TN}{TP + TN + FN + FP} \quad (5)$$

Research Questions: We raise the following three research questions to assess ToolPhet in terms of ① the effectiveness compared to other ML-based models, ② the comparison with a signature-based tool, and ③ the robustness to the unseen binary corpus.

- **RQ1.** How effective is ToolPhet compared to other ML baselines for a compilation toolchain prediction task (Section IX-B)?
- **RQ2.** How effective is ToolPhet compared to a signature-based tool? (Section IX-C)?
- **RQ3.** How robust is ToolPhet for completely unseen datasets? (Section IX-D)?

A. DATASET AND BASELINE MODELS

1) BINARY CORPUS

Table 3 briefly describes our dataset. We collect a variety of open-source programs written in Rust, Go, and Nim on GitHub [44] and Rosetta [45]. For GCC/clang compiler binaries, we utilize GNU Utilities [93] (*i.e.*, `binutils`, `coreutils`, and `findutils`). We build 1,683 executables in total on the x86-64 architecture. We follow a default option provided by each toolchain (*e.g.*, high optimization level, release mode), but leave debugging information for obtaining ground truth.

2) DATASET FOR MODEL GENERATION

We leverage BERT (unsupervised learning) into build a generic model by incorporating the entire 1,082,079 functions. This process results in the generation of 5,754 vocabularies (tokens) from those functions. We prepare additional datasets for building a toolchain classification model and a code similarity detection model (supervised learning). The former dataset simply contains a pair of (normalized function, label) where the label is an element of {Rust, Go, Nim, GCC, clang}. The latter dataset contains both similar and dissimilar (normalized) function pairs with a label of {True, False}. A similar pair (True) is chosen as the two functions with ① an identical symbol name from a different version of the same toolchain, and ② $T = 70$ or higher of a fuzzy hash value. Meanwhile, a dissimilar pair (False) is chosen as the two functions from a different toolchain or the same toolchain but different function bodies.

3) BASELINE MODELS

To demonstrate the effectiveness of ToolPhet, as a baseline of a machine learning approach, we develop two naïve neural-network-based models with RNN [47] and LSTM [48] (*i.e.*, multi-class classification). Additionally, we built two state-of-the-art compiler identification tools based on deep learning. First, we reproduce CNN-based O-glassesX [26] with our dataset for direct comparison. The original implementation of O-glassesX comes with two options; a model with disassembly and a model without it. We adopt the former setting for a fair comparison based on our observation of large performance degradation from the latter. Second, we prepare another model with NeuralCI [25]. The original implementation of NeuralCI comes with two options; CNN [94], [95] and RNN [47] structures. We adopt an RNN (*i.e.*, GRU [96]) model with Attention [97], [98], which shows the best performance. Note that we harness the function embeddings from SAFE [92] in comparison with the ToolPhet's similarity model because SAFE leverages Self-Attentive Neural Network to represent an assembly code. Finally, we exclude BinProv [6] because its source code has yet been opened at the time of writing.

B. EFFECTIVENESS OF TOOLPHET

To validate the effectiveness of the ToolPhet's TC model, we conducted performance comparisons with the above four baseline models. Table 4 summarizes the precision, recall, and F1 between ToolPhet and the baselines. Overall, ToolPhet achieves the best performance on macro-average with an F1 of 0.978. The RNN model shows the lowest F1 score, which we hypothesize that the RNN model suffers from a long dependency problem due to the length of input tokens (`Rust:190`, `Go:135`, `Nim:142`, `GCC:116`, `clang:141` on average). Meanwhile, the LSTM model moderately handles such a limitation with a memory cell. In case of the O-glassesX and NeuralCI models, they show similar or slightly lower performance (*e.g.*, F1 of 0.894 and

TABLE 4. Comparison of ToolPhet with other machine-learning-centered baseline models. Our toolchain classification model demonstrates the highest F1 of 0.978 on macro-average. P, R, and F1 denote a precision, recall and F1 score, respectively.

	Metric	TOOLPHET	NeuralCI	O-glassesX	LSTM	RNN
Rust	P	0.960	0.997	0.956	0.960	0.999
	R	0.999	0.999	0.966	0.993	0.609
	F1	0.979	**0.998**	0.961	0.976	0.757
Go	P	0.995	0.720	0.980	0.989	0.137
	R	0.997	0.999	0.955	0.747	0.708
	F1	**0.996**	0.837	0.967	0.851	0.230
Nim	P	0.986	0.981	0.754	0.589	0.062
	R	0.997	0.518	0.756	0.975	0.890
	F1	**0.996**	0.678	0.755	0.734	0.116
GCC	P	0.961	0.901	0.912	0.957	0.058
	R	0.976	0.856	0.906	0.980	0.357
	F1	**0.968**	0.878	0.909	0.968	0.100
clang	P	0.985	0.965	0.898	0.960	0.003
	R	0.928	0.892	0.904	0.979	0.099
	F1	0.956	0.927	0.901	**0.969**	0.006
Macro-Avg	P	0.977	0.913	0.896	0.899	0.252
	R	0.979	0.853	0.892	0.936	0.713
	F1	**0.978**	0.864	0.894	0.904	0.242

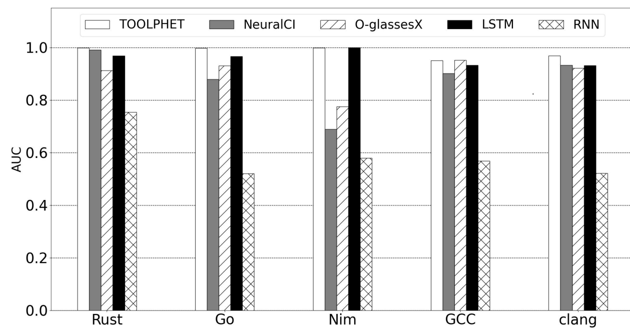

FIGURE 4. The performance comparison with AUC (Area Under the ROC Curve) between ToolPhet and other machine learning-centric baseline models. The AUC values for each model are presented for individual toolchains. Our toolchain classification model exhibits similar or higher AUC scores for each compilation toolchain compared to other baseline models.

0.864 on macro-average) than ToolPhet. Additionally, as can be seen from the results in Figure 4, ToolPhet shows better performance in overall AUC scores compared to the other models: 0.998, 0.879, 0.930, 0.966, and 0.529 for ToolPhet, NeuralCI, O-glassesX, LSTM, and RNN, respectively. This indicates that our model achieves a robust performance in inference accuracy compared to previously introduced models (*i.e.*, NeuralCI, O-glassesX) and naïve neural-network-based models (*i.e.*, RNN, LSTM) in the prior research.

> **Answer to RQ1.** Our empirical results demonstrate that the effectiveness of the TC model for a compiler toolchain inference task, outperforming existing baseline models.

C. COMPARISON WITH A SIGNATURE-BASED TOOL

In this section, we directly compare the effectiveness of the ToolPhet's TC model with that of DIE [46], one of the well-known signature-based tools.

TABLE 5. Performance comparison between the ToolPhet 's toolchain classification model and DIE [46] with the 300 executable binaries (*i.e.*, 20 binaries for each version). The table demonstrates the number of binaries that DIE and ToolPhet successfully predict their toolchain. DIE accurately infers Go, GCC, and clang binaries but fails to detect Rust and Nim cases. In contrast, our toolchain classification model makes accurate predictions for every case.

Toolchain	# of Total Binaries	DIE	TOOLPHET
Rust	80	0	80
Go	80	80	80
Nim	60	0	60
GCC	40	40	40
clang	40	40	40
Total (Ratio)	300 (100%)	160 (53.3%)	300 (100%)

1) DIE TOOL

DIE internally maintains a signature database on file information, thereby it can deterministically emit the property of a given file including a compiler toolchain (*e.g.*, compiler, linker). Additionally, DIE offers a feature to write one's own signature with a script in a flexible manner. However, the restriction of a signature-based tool is well-known: ① it cannot detect any (missing) information with the absence of a signature, and ② a signature database must be constantly maintained.

2) COMPARISON RESULTS

Table 5 summarizes the results of the toolchain prediction model in comparison with DIE. ToolPhet accurately infers the whole samples. While DIE successfully predicts all Go, GCC, and clang samples with appropriate signatures, it fails other cases (*e.g.*, Rust, Nim) due to the absence of corresponding signatures. Although the missing signatures could be further added, we stress that ToolPhet can reduce such maintenance overheads.

> **Answer to RQ2.** ToolPhet shows its effectiveness over a signature-based tool without the need of maintaining a signature database.

D. ROBUSTNESS OF TOOLPHET

This section expands our experiment to evaluate the robustness of ToolPhet with "`Unseen binaries`" because built-in functions across different versions from emerging compilation toolchains may vary because of frequent updates (Figure 2).

1) UNSEEN CORPUS

We deliberately generate the additional corpus of "`Unseen binaries`" that are distinct from both training and testing. We choose three versions from each toolchain: Rust: v1.46, v1.64, v1.68, Go: v1.13, v1.19, v1.20, and Nim: v1.2.18, v1.4.0, v1.6.12. We generate 20 binaries for each version, resulting in a total of 180 binaries.

TABLE 6. Performance comparison between the ToolPhet 's toolchain classification model and DIE [46] with the additionally generated unseen binary corpus. We discover that the model fails to predict some cases built with Nim. We confirm that the capability of DIE relies on the availability of signatures (*e.g.*, Rust and Nim binaries undetected).

Toolchain	# of Total Binaries	DIE	TOOLPHET
Rust	60	0	60
Go	60	60	60
Nim	60	0	30
Total (Ratio)	180 (100%)	60 (33.3%)	150 (88.9%)

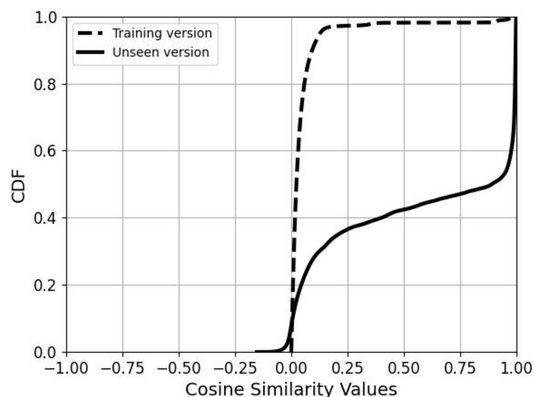

FIGURE 5. As Nim compilation involves both the frontend and backend of the GCC compiler, we investigate the similarity of common function bodies between training and unseen corpus. It turns out that the functions within unseen Nim binaries are similar to GCC functions (*e.g.*, around 50%), causing the confusion in our toolchain classification model.

2) RESULTS

Table 6 briefly displays the results of the toolchain prediction model in comparison with DIE. As one may expect, DIE accurately predicts the cases only when the signature is present in the database (*e.g.*, Go). Meanwhile, our TC model adequately infers a toolchain for Rust and Go samples but half of Nim samples (*i.e.*, 88.9%). It is noteworthy to mention that the samples are compiled with unseen versions while training. We further investigate faulty prediction cases where confusion arises in our TC model. Recall that Nim compilation utilizes both the frontend and the backend of the existing compiler (*e.g.*, GCC), which may cause misclassification. Figure 5 illustrates that the distribution of the common function embeddings (*i.e.*, pre-trainined BERT embedding) in the unseen corpus differs from the embeddings in the original training set.

3) CODE SIMILARITY DETECTION MODEL

We trained another downstream model to increase the robustness of ToolPhet, which can better deduce code semantics defining a code similarity detection downstream task. The model aims to detect a function that is semantically similar to the identified common functions in Section VII-B. Note that we utilize the BinShot [9] framework. As in Table 7, the CS model successfully predicts all unseen binaries with an average F1 of 0.978. Furthermore, we attempt to perform the code similarity task with another embedding from SAFE [92], which shows the effectiveness of ToolPhet 's embedding.

TABLE 7. Experimental results for the ToolPhet's code similarity (CS) model with the unseen dataset in Table 6. The model predicts all binaries with a high F1. We believe that both classification and similarity models can be complementary for accurate prediction. Furthermore, to demonstrate how function embeddings are well tuned for a code similarity task, we compare performance of ToolPhet with that of SAFE's embeddings (Note that SAFE merely provides a function-unit vector). P, R, and F1 denote a precision, recall, and F1 score, respectively.

Toolchain	Metric	CS model (BERT embedding)	CS model (SAFE embedding)
Rust	P	0.904	0.456
	R	0.990	0.622
	F1	**0.971**	0.565
Go	P	0.894	0.636
	R	0.990	0.670
	F1	**0.949**	0.613
Nim	P	0.918	0.845
	R	0.999	0.376
	F1	**0.957**	0.519
Macro-Average	P	0.959	0.626
	R	0.998	0.571
	F1	**0.978**	0.561

Answer to RQ3. Our experiment with an unseen binary corpus shows that ToolPhet may misclassify certain cases with our TC model. The CS model for inferring code semantic difference can address this issue.

X. DISCUSSION AND LIMITATIONS

A. BINARY REPRESENTATIVENESS

Although the popularity of emerging toolchains has been increasing, the number of applications with them in the wild is still far less than those with previous toolchains (*e.g.*, GCC, clang). In this regard, prior study of toolchain provenance focuses on mostly GCC and clang [4], [4], [6], [25], [26], [28] with a popular dataset like GNU utilities [93] and OpenSSL [99]. Our dataset collected from Github and Rosetta would be insufficient, which may be difficult to generalize our model at this point.

B. FALSE POSITIVE RATE WITH A CODE SIMILARITY MODEL

Our observation shows that a ToolPhet's code similarity detection model might be indecisive for a certain case. Recall that we cannot pinpoint common functions that fingerprint a particular toolchain in the beginning because they are spread out in a binary, choosing part of functions at random. This inevitably contains a user-defined function that may be similar to a list of functions of our interest for comparison, highly causing a false positive rate. Furthermore, the list might be ineffective if a representative function would have been considerably updated as shown in Figure 2. We leave such limitations to be addressed in our future research.

C. COMPILER TOOLCHAIN PROVENANCE

Previous work on compiler toolchain provenance has typically involved the identification of a toolchain, an optimization level, or a compiler version. This work merely focuses on a toolchain itself because emerging toolchains sometimes do not offer an optimization level (*e.g.*, O0-O3, Os) but an easily identifiable mode (*e.g.*, debugging, release). We leave the recognition of an emerging compiler toolchain version as part of our future work.

XI. CONCLUSION

In this paper, we introduce ToolPhet, a BERT-assisted end-to-end system for inferring the provenance of an emerging compiler toolchain (*e.g.*, Rust, Go, Nim) as well as GCC and clang. We first train a generic model with executable binaries built from the emerging toolchain, then fine-tune the model for a downstream task in two different ways; *i.e.*, toolchain recognition with a toolchain classification model, and with a code similarity detection model. Our experimental results show that ToolPhet outperforms existing approaches including both a signature-based one (*e.g.*, DIE) and neural-network-based ones (*e.g.*, CNN, RNN, LSTM).

ACKNOWLEDGMENT

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the sponsor's views.

REFERENCES

[1] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proc. 23rd USENIX Conf. Secur. Symp. (SEC)*. Berkeley, CA, USA; USENIX Association, 2014, pp. 845–860.

[2] A. Di Federico, M. Payer, and G. Agosta, "Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries," in *Proc. 26th Int. Conf. Compiler Construct.* New York, NY, USA: Association for Computing, Feb. 2017, pp. 131–141.

[3] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Conf. Secur. Symp. (SEC)*. Berkeley, CA, USA; USENIX Association, 2015, pp. 611–626.

[4] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association for Computing, Jul. 2011, pp. 100–110.

[5] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, May 2010, pp. 21–28.

[6] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "BinProv: Binary code provenance identification without disassembly," in *Proc. 25th Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*. New York, NY, USA: Association for Computing, Oct. 2022, pp. 350–363.

[7] Z. Luo, T. Hou, X. Zhou, H. Zeng, and Z. Lu, "Binary code similarity detection through LSTM and Siamese neural network," *ICST Trans. Secur. Saf.*, vol. 8, no. 29, Nov. 2021, Art. no. 170956.

[8] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, "BinDeep: A deep learning approach to binary code similarity detection," *Expert Syst. Appl.*, vol. 168, Apr. 2021, Art. no. 114348. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420310332

[9] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with BERT-based transferable similarity learning," in *Proc. 38th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Austin, TX, USA. New York, NY, USA: Association for Computing Machinery, 2022, pp. 361–374.

[10] R. A. Erinfolami and A. Prakash, "DeClassifier: Class-inheritance inference engine for optimized C++ binaries," in *Proc. ACM Asia Conf. Comput. Commun. Secur. (ASIACCS)*. New York, NY, USA: Association for Computing, Jul. 2019, pp. 28–40.

[11] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing, Oct. 2018, pp. 1667–1680.

[12] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*. New York, NY, USA: Association for Computing, Dec. 2020, pp. 373–385.

[13] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Lett.*, vol. 34, no. 3, pp. 103–104, Nov. 2014.

[14] ZDNet. (2022). *Linus Torvalds: Rust Will Go Into Linux 6.1*. [Online]. Available: https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[15] J. Meyerson, "The go programming language," *IEEE Softw.*, vol. 31, no. 5, p. 104, Sep. 2014.

[16] A. Rumpf. (2022). *A Statically Typed Compiled Systems Programming Language*. [Online]. Available: https://nim-lang.org/

[17] P. Jansen. (2022). *Tiobe Index—The Go Programming Language*. [Online]. Available: https://www.tiobe.com/tiobe-index/go/

[18] P. Jansen. (2022). *Tiobe Index*. [Online]. Available: https://www.tiobe.com/tiobe-index/

[19] ZDNet. (2021). *Malware Rewritten in the Rust Programming Language*. [Online]. Available: https://www.zdnet.com/article/this-malware-has-been-rewritten-in-the-rust-programming-language-to-make-it-harder-to-spot/

[20] ZDNet. (2021). *Go Malware is Now Common, Having Been Adopted By Both APTs and E-crime Groups*. ZDNet. [Online]. Available: https://www.zdnet.com/article/go-malware-is-now-common-having-been-adopted-by-both-apts-and-e-crime-groups/

[21] Hacker-News. (2021). *Malware Written in Nim Programming Language*. [Online]. Available: https://thehackernews.com/2021/03/researchers-spotted-malware-written-in.html

[22] Trend Micro. (2022). *Agenda Ransomware Uses Rust To Target More Vital Industries*. [Online]. Available: https://www.trendmicro.com/en_us/research/22/l/agenda-ransomware-uses-rust-to-target-more-vital-industries.html

[23] Dragos. (2020). *EKANS Ransomware and ICS Operations*. [Online]. Available: https://www.dragos.com/blog/industry-news/ekans-ransomware-and-ics-operations/

[24] R. Tay and J. Salvio. (2022). *New IceXLoader 3.0 - Developers Warm Up To Nim*. [Online]. Available: https://www.fortinet.com/blog/threat-research/new-icexloader-3-0-developers-warm-up-to-nim

[25] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen, and D. Wu, "Fine-grained compiler identification with sequence-oriented neural modeling," *IEEE Access*, vol. 9, pp. 49160–49175, 2021.

[26] Y. Otsubo, A. Otsuka, M. Mimura, T. Sakaki, and H. Ukegawa, "O-GlassesX: Compiler provenance recovery with attention mechanism from a short code fragment," in *Proc. Workshop Binary Anal. Res.*, 2020.

[27] T. Benoit, J.-Y. Marion, and S. Bardin, "Binary level toolchain provenance identification with graph neural networks," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2021, pp. 131–141.

[28] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "BinComp: A stratified approach to compiler provenance attribution," *Digit. Invest.*, vol. 14, pp. S146–S155, Aug. 2015.

[29] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Softw., Pract. Exp.*, vol. 25, no. 7, pp. 811–829, Jul. 1995.

[30] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog, "Fileprints: Identifying file types by n-gram analysis," in *Proc. 6th Annu. IEEE Syst., Man Cybern. (SMC) Inf. Assurance Workshop*, Jun. 2005, pp. 64–71.

[31] Hex-Rays. (2022). *IDA Pro Disassembler*. [Online]. Available: https://www.hex-rays.com/products/ida/

[32] National-Security-Agency. (2022). *Ghidra Software Reverse Engineering Framework*. [Online]. Available: https://ghidra-sre.org/

[33] L. Chen, Z. He, H. Wu, F. Xu, Y. Qian, and B. Mao, "DIComP: Lightweight data-driven inference of binary compiler provenance with high accuracy," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 112–122.

[34] M. Ismail, Y. Lin, D. Han, and D. Gao, "BinAlign: Alignment padding based compiler provenance recovery," in *Information Security and Privacy*. Cham, Switzerland: Springer, 2023, pp. 609–629.

[35] Y. Du, O. Alrawi, K. Snow, M. Antonakakis, and F. Monrose, "Improving security tasks using compiler provenance information recovered at the binary-level," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing, Nov. 2023, pp. 2695–2709.

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.* Minneapolis, MI, USA: Association for Computational Linguistics, vol. 1, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[37] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates, 2017, pp. 3149–3157.

[38] GCC. (1987). *GCC, The GNU Compiler Collection*. [Online]. Available: https://gcc.gnu.org/

[39] LLVM. (2003). *The LLVM Compiler Infrastructure*. [Online]. Available: https://llvm.org/

[40] T. Wolf et al., "Huggingface's transformers: State-of-the-art natural language processing," 2019, *arXiv:1910.03771*.

[41] Huanghonggit. (2019). *BERT MLM with PyTorch*. [Online]. Available: https://github.com/huanghonggit/Mask-Language-Model

[42] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 1–38, Apr. 2021.

[43] Z. Liu, "Binary code similarity detection," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1056–1060.

[44] GitHub. (2022). *GitHub*. [Online]. Available: https://github.com/

[45] RosettaCode-Contributors. (2022). *Rosetta-Code*. [Online]. Available: https://rosettacode.org/w/index.php?title=Rosetta_Code&oldid=322370

[46] Detect-It-Easy. (2022). *Program for Determining Types of Files for Windows, Linux and MacOS*. [Online]. Available: https://github.com/horsicq/Detect-It-Easy/

[47] J. Elman, "Finding structure in time," *Cognit. Sci.*, vol. 14, no. 2, pp. 179–211, Jun. 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/036402139090002E

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[49] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybern.*, vol. 36, no. 4, pp. 193–202, Apr. 1980.

[50] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016, *arXiv:1409.0473*.

[51] A. Adhikari and P. A. Kulkarni, "Using the strings metadata to detect the source language of the binary," in *Proc. Int. Conf. Innov. Comput. Res. (ICR)*, K. Daimi and A. A. Sadoon, Eds. Cham, Switzerland: Springer, 2022, pp. 190–200.

[52] B. W. Kernighan and D. M. Ritchie, *The C Programming* (Prentice Hall Professional Technical Reference), 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1988.

[53] B. Stroustrup. (1985). *The Programming Language C++*. [Online]. Available: https://isocpp.org/

[54] A. B. Cox and T. Love. (2016). *Objective-C*. [Online]. Available: https://developer.apple.com/library/archive/navigation/

[55] G. V. Rossum and F. L Drake Jr. (2022). *Python Reference Manual*. [Online]. Available: https://www.python.org/psf-landing/

[56] Y. Matsumoto. (2022). *Ruby Programming Language*. [Online]. Available: https://www.ruby-lang.org/

[57] R. I. R. Gentleman. (2022). *The R Project for Statistical Computing*. [Online]. Available: https://www.r-project.org/

[58] L. Wall. (2022). *The Perl Programming Language*. [Online]. Available: https://www.perl.org/

[59] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. (2022). *The Programming Language Lua*. [Online]. Available: https://www.lua.org/

[60] J. Gosling. (2022). *The Programming Language Java*. [Online]. Available: https://www.java.com/

[61] Microsoft Corporation. (2022). *The Programming Language Java*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/

[62] ECMA International Mozilla Foundation, Netscape Communications. (2022). *Javascript-MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/

[63] J. Gosling. (2022). *Java Virtual Machine*. Sun Microsystems. [Online]. Available: https://www.oracle.com/it-infrastructure/

[64] The Chromium Projects. (2022). *Javascript Engine V8*. [Online]. Available: https://v8.dev/

[65] X. Li, Y. Qu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing, Nov. 2021, pp. 3236–3251.

[66] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing, Oct. 2017, pp. 363–376.

[67] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

[68] P. Cerda, G. Varoquaux, and B. Kégl, "Similarity encoding for learning with dirty categorical variables," 2018, *arXiv:1806.00979*.

[69] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[70] H. Koo, S. Park, D. Choi, and T. Kim, "Binary code representation with well-balanced instruction normalization," *IEEE Access*, vol. 11, pp. 29183–29198, 2023.

[71] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Invest.*, vol. 3, pp. 91–97, Sep. 2006.

[72] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization options from binary code using deep learning approaches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 232–242.

[73] Snaker, Qwerton, Jibz, and XineohP. (2022). *PE iDentifier*. [Online]. Available: https://www.aldeid.com/wiki/PEiD

[74] Microsoft. (2022). *PE Format*. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[75] Unix. (2022). *ELF(Executable and Linkable Format) Format*. [Online]. Available: https://unix.org/

[76] Apple Inc., Carnegie Mellon University. (2022). *Mach Object File Format*. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html

[77] Intel. (2002). *Intel oneAPI DPC++/C++ Compiler*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

[78] Microsoft. (1997). *Visual Studio*. [Online]. Available: https://visualstudio.microsoft.com/

[79] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2013, pp. 48–62.

[80] NPM. (2022). *Node Package Manager for JavaScript's Runtime Node.js*. [Online]. Available: https://www.npmjs.com/

[81] Crates. (2022). *The Rust Community's Crate Registry*. [Online]. Available: https://crates.io/

[82] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. W. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Comput. Syst.*, vol. 8, pp. 221–254, Jan. 1995.

[83] LLVM. (2022). *LLVM-Based Go Compiler*. [Online]. Available: https://go.googlesource.com/gollvm/

[84] Nimlang. (2022). *Official Github Repository for Nimlang*. [Online]. Available: https://github.com/nim-lang/Nim

[85] Oracle. (2022). *Java Native Interface Specification*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

[86] Microsoft. (2022). *Platform Invoke (P/Invoke)*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

[87] A. Rumpf and Z. Karadjov. (2022). *Nim Manual*. [Online]. Available: https://nim-lang.org/docs/manual.html#pragmas-compilation-option-pragmas

[88] J. Sieka. (2022). *LLVM-Based Compiler for the Nim Language*. [Online]. Available: https://github.com/arnetheduck/nlvm

[89] Rust-Contributors. (2018). *Rust Symbol Name Mangling-V0*. [Online]. Available: https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html
[90] Trend Micro. (2022). *Closing the Door: DeadBolt Ransomware Locks Out Vendors With Multitiered Extortion Scheme*. [Online]. Available: https://www.trendmicro.com/en_us/research/22/f/closing-the-door-deadbolt-ransomware-locks-out-vendors-with-mult.html
[91] S. C. A. Paszke, S. Gross, and G. Chanan. (2019). *Open Source Machine Learning Framework*. [Online]. Available: https://pytorch.org/
[92] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni, "Safe: Self-attentive function embeddings for binary similarity," in *Proc. 16th Conf. Detection Intrusions Malware Vulnerability Assessment*, 2019.
[93] GNUisance. (2022). *GNU Operating System*. [Online]. Available: https://www.gnu.org/
[94] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015, *arXiv:1511.08458*.
[95] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *Proc. Int. Conf. Eng. Technol. (ICET)*, Aug. 2017, pp. 1–6.
[96] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*.
[97] A. Galassi, M. Lippi, and P. Torroni, "Attention in natural language processing," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 10, pp. 4291–4308, Oct. 2021.
[98] L. Dong and M. Lapata, "Language to logical form with neural attention," 2016, *arXiv:1601.01280*.
[99] OpenSSL. (2022). *Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org

HOHYEON JANG received the B.S. degree from Baekseok University, in 2020. He is currently pursuing the master's degree with the Department of Computer Science and Engineering, College of Computing, Sungkyunkwan University. He is a member of the Security with AI (SecAI) Laboratory. His research interests include reverse engineering, malware, and binary analysis using AI.

NOZIMA MURODOVA received the bachelor's degree from the Department of Computer Science, Inha University, Tashkent. She is currently pursuing the degree with the Department of Computer Science and Engineering, College of Computing, Sungkyunkwan University. She is with the Security with AI (SecAI) Laboratory under the supervision of Prof. Hyungjoon Koo. Her main research interest includes software security with artificial intelligence.

HYUNGJOON KOO received the Ph.D. degree in computer science from Stony Brook University, in 2019. He is currently an Assistant Professor with the Department of Computer Science and Engineering, College of Computing, Sungkyunkwan University (SKKU). Before joining SKKU, he was a Postdoctoral Researcher with the Security with AI (SecAI) Laboratory, Georgia Institute of Technology. He is also leading the SecAI Laboratory. His research interests include software security, network security, binary analysis and protection, and security leveraging artificial intelligence.