What Makefile? Detecting Compiler Information Without Source Using The Code Property Graph

Sean Deaton
Blue Star and Bogart Associates
Columbia Maryland, USA
sean.deaton@proton.me

***Abstract*—Users frequently lack access to the underlying source code and build artifacts of the programs they use. Without access, uncovering information about programs, such as compiler information or security properties, becomes a difficult task. Various methods exist for static analysis testing on source code languages, but few tools work solely with the executable machine code. This paper proposes constructing the code property graph from a program's lifted machine code to observe structural differences between other executables. We implement our approach with the Binary Ninja Intermediate Language (BNIL) and the `graph2vec` neural embedding framework to create embedded representations of the graphical properties of the program. Downstream applications, such as supervised machine learning, can then analyze these representations. We demonstrate the effectiveness of our approach by training a supervised random forest classifier on the embedded graphs to determine, at the function level, which compiler, `clang` or `gcc`, created the executable the function belongs to. Our results achieved an accuracy of 100% across our testing set of 25,600 samples.**

***Index Terms*—compiler, code property graph, static analysis, Binary Ninja, graph2vec**

I. Introduction

Many tools exist to detect information about a program statically. Termed Static Application Security Testing (SAST), these applications parse the source code in some way, without running it, to arrive at a conclusion about the program. In many cases, this conclusion is about the security of a program. Some compilers implement static application testing to warn users of potential errors and optimize the code. In other cases, companies sell SAST tools commercially to parse source code for vulnerabilities. [1]

Except for open-source software, only on rare occasions is source code provided to the end-user for inspection alongside the executable program. Without access, users must trust that the original software developers correctly implemented the program and did not introduce any security flaws. Alternatively, users can use a disassembler or decompiler to inspect the program. However, this process can be challenging as decompilers can never fully reconstruct the source code as they typically optimize for binary size or execution speed instead of reproducibility [2].

Over the years, decompilers have improved in reconstructing accurate C syntax (which we call pseudo-C since there are usually no guarantees about the ability to recompile it), as noted by Liu and Wang in [3]. However, they still largely lack accurate type recovery and the ability to derive meaning from heavily optimized routines [2]. While there have been attempts to improve these facets when returning machine code to the original source [2] [4], many instead focus reversing efforts on intermediate languages given that they can often be more meaningful and expressive [3].

Researchers have recently focused on automatically inspecting compiled executables for software vulnerabilities or other artifacts instead of manually reviewing decompiled programs. In [5], the authors demonstrate the use of IDA Pro in disassembling compiled machine code to detect suspicious API calls. Yuan and Ding introduced a method for detecting stack-based buffer overflows without source code in [6].

One particularly effective tool for automated analysis is Joern [7]. Joern is a Scala program that creates the code property graph, discussed in Section II, for each function in a program and allows users to query specific attributes about its execution. A researcher could, for example, search for all blocks of code reachable from a given parent block or examine all statements where the input to a function is read. Joern primarily supports source code languages such as C and Java, though it does support `x86` and `amd64` executables by decompiling the executable with Ghidra and parsing the pseudo-C syntax [7].

Using the code property graph, especially those created by Joern, for downstream applications is not a novel idea. For example, both Xiaomeng et al. in [8] and Xu et al. in [9] trained deep learning models on the code property graph generated from vulnerable and not-vulnerable source code functions. In [10], Haojie et al. further extended the code property graph constructed by Joern with the function call graph to detect cross-function vulnerabilities when source code is available.

However, most of these applications require either the availability of source code or the output of Ghidra's decompilation as pseudo-C. Given that source code is not always provided and decompilation is not exact, there exists a gap in many of these models for real-world applications. We propose that lifting a program's machine code to an intermediate language and then creating the code property graph will fill this gap.

Funding provided by Blue Star and Bogart Associates under the Internal Research and Development program.

978-1-6654-7408-5/22/$31.00 ©2022 IEEE
DOI 10.1109/TPS-ISA56441.2022.00039

A. Motivation

Manual reverse engineering is a tedious task. Although nothing can replace inspecting a program in a decompiler, we seek to automatically uncover as much information as early as possible before a reverse engineer ever looks at the decompiled output. Early access to information would enable reverse engineers to hone in on particular functions or subroutines of a program to search for specific artifacts.

In this paper, we introduce a workflow that performs analysis on executables without source code to derive information from the graphical structure of the program automatically. We do this by 1) lifting the syntax of each function in a program to an intermediate language to extract as much meaningful information as possible; 2) creating a graphical representation of the intermediate language by constructing the code property graph for each function; 3) training an unsupervised graph neural network on the whole structure of the graph to yield the graph embeddings. The output can yield information about the structural similarities or differences between functions. We test our approach by detecting the graphical differences introduced in each function by two different compilers - `clang` and `gcc`.

B. Contributions

We make the following contributions to static analysis and security testing:

- Implementation of the code property graph from an intermediate language. We demonstrate that it is possible and meaningful to create the code property graph on executables without using source code or decompiling the executable back to some source code language (for example, pseudo-C) by instead using an intermediate language.
- Detection of the compiler used to create the executable. We show that the embedded features of the code property graph can be given to a machine learning model to accurately predict which compiler (`clang` versus `gcc`) created the executable.

II. Background

A. Code Property Graph

Much work has been done on analyzing source code and its syntax as a formal language. However, this can naturally lose semantic meaning, as program and data dependencies can be difficult to retain [11]. Instead, researchers have developed methods to view a program as a graph to retain its structure and semantics. Viewing the program in this manner avoids problems with other methods, such as natural language processing, that may rely on the sequence or frequency of program statements.

Compilers use the abstract syntax tree to represent and nest program statements - it provides the program's structure. The tree's innermost nodes consist of operators, and the leaf nodes contain the operands [12]. The typical graph view found in most decompilers, the control flow graph, was first described by Alan Frances in [13]. The control flow graph depicts the different basic blocks a program can visit based on conditional properties, the quintessential example being an `if` statement. Ferrante et al. later created the program dependence graph in [14] to show data and control dependencies explicitly. Note that the program dependence graph does not maintain the order of execution of program statements. Instead, the graph depicts which statements must have been executed before reaching another statement.

Most recently, Yamaguchi et al. in [12] developed the code property graph. This graph combines the abstract syntax tree, control flow graph, and program dependence graph to gain the advantages of each one. The abstract syntax tree provides the program's statements; the control flow graph describes the order in which the statements execute; the program dependence graph preserves the dependencies needed to reach each statement. The code property graph has since been used to discover vulnerabilities in programs such as the Linux kernel [12] and validate those included in the National Institute of Standards and Technology's Software Assurance Reference Dataset [15].

B. Graphical Embeddings

Although we can represent programs in this graph-structured manner, it is of little use to many downstream analytical tasks in its raw state. Many of these tasks, including classification and clustering, require fixed-length feature vectors [16]. However, fixed-length vectors are naturally an ill fit for graphs, which can have any number of nodes and edges.

Various methods have been developed to learn the embeddings of graphical structures and emit the result as a fixed-length feature vector compatible with machine learning models. `node2vec` was developed to understand and preserve the neighborhood of nodes [17]. Similarly, Deep Graph Kernels were proposed in [18] to learn the latent representation of graphs by understanding the paths between nodes. `subgraph2vec` was proposed in [19] to encode rooted subgraphs for use in downstream classifiers such as convolutional neural networks and support vector machines.

These models, however, cannot learn the representation of entire graphs. `graph2vec` was proposed by Narayanan et al. in [16] to solve this issue and learn the embeddings for entire graphs.

C. Intermediate Languages

Intermediate languages position themselves in between source code and machine code. In contrast to assembly, they provide a richer notation that allows downstream applications, such as compilers, to optimize the program. Generally, they are extremely expressive, strongly typed, and provide fewer instructions than assembly.[1]

One example of the expressiveness of intermediate languages comes from implementing a property known as static single-assignment (SSA) form. Developed by Cytron et al. in [22], SSA form requires that each variable only be assigned

[1]Compare Intel's 981 x86-64 mnemonics [20] to Ghidra's 72 P-Code operations [21].

once. Subsequent uses that involve reassignment of the same variable generate new version numbers. Version numbers help clarify the possible contents of a variable at a particular site. Several of the following intermediate languages implement SSA form.

1) LLVM: Many static analysis tools operate on the LLVM intermediate language [23] [24] [25]. The native support from Clang and ubiquitous tooling make it popular for security researchers to perform analysis. However, LLVM is typically emitted from source code. Lifting machine code to LLVM is difficult because of the strongly typed memory model [26], though there are tools that do so with several caveats [27] [28].

2) P-Code: P-Code is an intermediate language implemented by the National Security Agency's open source reverse engineering framework, Ghidra. P-Code lifts machine code with the goal being to emit the pseudo-C code shown by the decompiler. It supports most popular architectures, such as `amd64`, `powerpc`, and `mips`. Community members have implemented other, more niche architectures given Ghidra's open-source status.

P-Code is comprised of three main elements: `p-code operations` that change the processor state and mimic machine code instructions; an `address space` which represents a sequence of bytes that can be read or written to by `p-code operations`; and `varnodes` that represent registers or memory address locations inside of an `address space`. [29]

3) Microcode: The Interactive Disassembler (IDA) by Hex-Rays lifts machine code to their intermediate language, Microcode. Microcode is a very mature intermediate language by virtue of being included with IDA and given its 24 years of development [30].

There are 72 Microcode instructions, each in the form: `opcode left, right, destination` where the latter three elements represent the operands of the `opcode`. An `opcode` can represent different instructions, such as shifting values, generating conditions, and moving values.

Each `opcode` performs only one operation without side effects which can simplify meaning. Consequently, however, each assembly instruction will map to several `opcodes`. This, along with its verbose syntax, can make it difficult for reverse engineers to understand.[2]

Instead of reading the Microcode directly, the syntax maps to a `ctree` (similar to an abstract syntax tree) that cleans up the output. The intent is for the `ctree` to naturally map to pseudo-C syntax, which many reverse engineers find desirable [30].

4) BNIL: The Binary Ninja Intermediate Language (BNIL) is a collection of intermediate languages developed by Vector 35 for their reverse engineering framework, Binary Ninja. It supports a number of different architectures including `arm`, `x86`, `amd64`, and `powerpc`. Developers can add support for additional languages via architecture plugins.

[2]This is intentional and noted by the creator of the language. However, improving readability is one area Hex-Rays is working to improve [30].

BNIL is made up of four key abstraction languages: the Lifted Intermediate Language (Lifted IL), Low Level Intermediate Language (LLIL), Medium Level Intermediate Language (MLIL), and High Level Intermediate Language (HLIL). Lifted IL literally translates instructions into architecture agnostic semantic instructions. LLIL takes those same instructions to create conditionals from instructions that operate on flags. MLIL introduces variables and their type information based on register and memory accesses. MLIL generates function call sites with their parameters based on the detected calling convention. HLIL abstracts MLIL by adding control flow information and removing code that it determines the program cannot reach (dead code elimination) [31].

5) Other: A myriad of other intermediate languages also exist. BAP (Binary Analysis Platform) by Carnegie Melon University is used with popular downstream tools such as Mayhem and CWE Checker. [32] The VEX intermediate language is used in `angr` for concolic execution [33]. The Reverse Engineering Intermediate Language (REIL) is another well-defined intermediate language but lacks widespread adoption in downstream applications and reverse engineering workflows [34].

III. Methods

Figure 1 presents an overview of our classification process. We breakdown this approach into four phases: 1) preprocessing of the executables into Binary Ninja database files; 2) construction of the code property graph for each function in each of the Binary Ninja database files, saved as GraphML files [35]; 3) fitting of the `graph2vec` unsupervised machine learning model with the GraphML files; 4) training and classification using the supervised Random Forest Classification machine learning model.

1) Intermediate Language Selection: We selected Binary Ninja's Intermediate Language (BNIL) as our intermediate language of choice for the following reasons:

- Binary Ninja uses BNIL and exposes the API through a Python console in the GUI. Native access inside the decompiler makes it ideal for practical reverse engineering compared to representations such as BAP or VEX, which are primarily used by downstream tools like `angr`.
- The API is open source.
- Researchers and developers can add support for other architectures via architecture plugins.
- The commercial license allows interfacing with BNIL without the GUI via exposed Python bindings.

In contrast, we dismissed Hex-Ray's Microcode because, although it does include Python bindings, we found the documentation surrounding the Microcode API to be insubstantial. Furthermore, the cost was prohibitive at $1975 for the professional license. [36]. We did not consider the cheaper home license because it provides access to only a single architecture and sends back markup information to Hex-Rays.

Ghidra does provide a well-documented API for interacting with P-Code; however, the bindings can only be accessed with an included `analyzeHeadless` executable and not

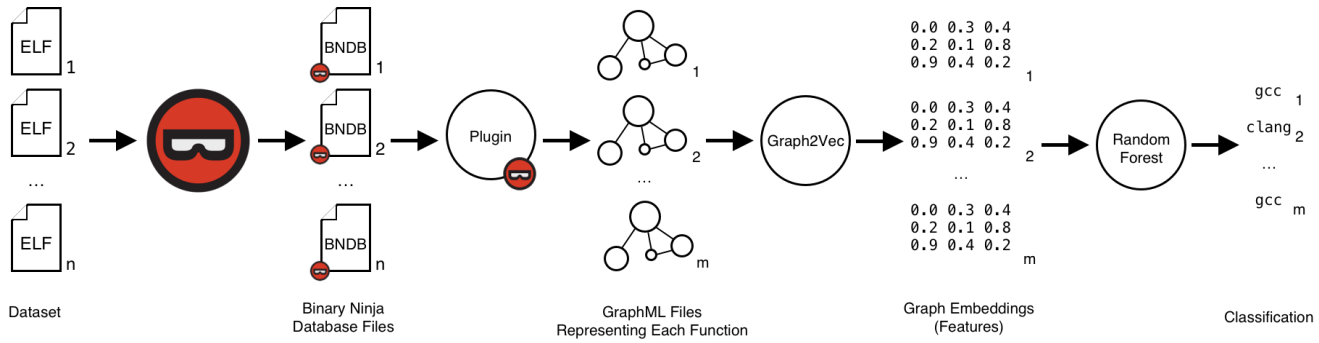


Fig. 1. Compiler classification workflow.

directly imported as with Binary Ninja. Furthermore, the languages supported, Java and Jython 2.7, are incompatible with popular machine learning frameworks like sklearn, which are implemented in Python. [37]

Given that Ghidra and IDA provide access to the underlying abstract syntax tree (or `ctree` in IDA's case), it would be trivial to implement our methods with either intermediate language.

2) Dataset: Our downstream model performs supervised learning, meaning we must have executables with binary labels corresponding to the compiler used to create each program - `clang` or `gcc`. For this dataset, we used a subset of the executables compiled in [38] and made initially available on Zenodo at [39]. The programs in the dataset consist of shared libraries compiled from the *Linux from Scratch* book,[3] version 9.1-systemd published March 1st, 2020.

In particular, we selected all of the programs compiled for the `amd64` architecture without any optimizations (the `-o0` flag) for both `clang` and `gcc`. This resulted in 1,937 and 1,938 samples, respectively. All the executables between the two compiler sets were identical save for `libelf-0.178.so`, which was compiled for `gcc` and not `clang`. We removed this file from the dataset so that the sets were identical. We also removed any `.mod` and `.a` files to leave only `ELF` files. This resulted in 2,972 samples split equally between `clang` and `gcc`. Shared libraries were used in lieu of linked executables to avoid any optimizations performed on the executable at link time [39]. We selected this architecture and optimization level to limit the number of independent variables; further experiments could test our approach to detecting other compiler combinations.

The executables were relatively small, with the largest being `clang-10` at 207.7 MB and the smallest being `all_video.module` at 492 bytes. The median size was 30.8 KB and the mean 2.6 MB.

A. Preprocessing

Using the Binary Ninja headless API made available with the commercial license, our preprocessor iterated over our dataset to create Binary Ninja database files (abbreviated as `.bndb` files). Each `.bndb` contains the analysis results that Binary Ninja performs on the executable. Each database file includes any found symbols, function boundaries, and the disassembly results for the entire executable. The `.bndb` files provide access to the BNIL used in the following stages. The total number of `.bndb` files are equal to our dataset's total number of executables.

B. Code Property Graph Creation

Following the prepossessing performed by Binary Ninja, we pass each `.bndb` to a Binary Ninja plugin that we created to construct the code property graph for each function in the executable. We consider only functions that are not thunks. We define a thunk-ed function as one that invokes a tail call[4] within two HLIL operations.[5]

We construct the code property graph by first creating the abstract syntax tree. For this, we consider Binary Ninja's HLIL syntax in SSA form. To build the abstract syntax tree, we first have to consider the main entry point of the function. To do so, we consider the first basic block without incoming edges. This represents the entry point of the function. When all of the functions' basic blocks have incoming edges, we check if any of the blocks have incoming back edges. Back edges can occur when a function begins with a `for` or `while` loop, where the conditional check of the loop dominates the inner execution of the loop. Because Binary Ninja's HLIL eliminates dead code branches, we did not observe any functions that did not meet one of these conditions. We present the algorithm in Listing 1.

We then consider each HLIL expression in the basic block. We check the type of expression and handle it appropriately. Notably, if the expression represents an operation and thus has operands, we first add the operation to the abstract syntax tree. We then recursively descend on the operands, evaluate them, and add them as children to the operation. If the expression is

[3]http://www.linuxfromscratch.org/lfs/index.html

[4]A tail call usually indicates that the function jumps to a separate subroutine, often within another linked executable.

[5]After our experiments, we implemented a slightly different approach using LLIL which Vector 35 merged into the Binary Ninja API in pull request #3343. [40]

```
def get_starting_block(
            function: Function)
             -> BasicBlock:
    # If the block has no incoming edge
    # it must be reached by the start
    # of the function.
    for block in function.basic_blocks:
        if not block.incoming_edges:
            return block
    # Otherwise, get the source of the
    # first block's back edge.
    for edge in function.basic_blocks[0]
                                  .edges:
        if edge.back_edge:
            return edge.source
```

Listing 1: Visiting the starting block of each function.

```
def visit_expr(block: BasicBlock,
        expr: ExpressionType) -> Node:
    children = []
    if isinstance(expr, Constant):
        # Lookup the value of the constant.
    elif hasattr(expr, 'operands'):
        for operand in expr.operands:
            children.append(
                visit_expr(operands))
    elif hasattr(expr, 'name'):
        children.append(
            visit_expr(expr.name)
    ...
    # At the last expression in the block.
    # Check if there are outgoing edges.
    if expr == block[-1]:
        # Visit the target block
        # of each outgoing edge.

    return Node(expr, children)
```

Listing 2: Visiting the expressions in each basic block.

a constant, we can return the literal value of the constant as a string. Other cases are possible, such as SSA version numbers and expression names.

Constructing the control flow graph and program dependence graph is more straightforward and does not create new nodes; the abstract syntax tree is the only graph that adds additional nodes [12]. First, we consider all of the functions' basic blocks to construct the control flow graph. Then, for each block, we examine the outgoing edges of the block. Of those outgoing blocks, we add an edge onto the control flow graph connecting the source block's last instruction to the first instruction of the target block. We then iterate down all the blocks' expressions and add an edge between them.

Binary Ninja exposes each function's dominance frontier, allowing us to construct the program dependence graph. First, we examine each basic block and observe which children it dominates, if any. We then add an edge between the last instruction of the dominating block and the first instruction of the child block. Note that this does not explicitly add edges between data dependencies in the function - it only shows the program dependencies. However, because we consider only the SSA representations of BNIL, we implicitly gain the data dependencies because no variable is ever assigned twice.

We combine all three graphs to form the code property graph as described in [12]. Since each node can have multiple parallel edges, each graph must be in the form of a multi-digraph. Finally, we save the results as GraphML files. The number of GraphML files equals the total number of functions found in all of the `.bndb` files.

C. *graph2vec*

We created a `graph2vec` model instance as described in [16] and implemented in the Karate Club [41] Python library [42]. The model takes all of the GraphML files and reads them in as NetworkX graphs [43]. We then fit the model with all of the NetworkX graphs. The model emits the whole graph embeddings for each function as a 2-D array of continuous values from -1.0 to 1.0 floating point across 128 dimensions. Graphs that are similar in the embedding space also share similar structural patterns, which is the basis of this experiment [41].

D. *Training and Classification*

We created a Random Forest Classifier within scikit-learn [37] with default hyperparameters. The graph embeddings from the previous stage served as the feature vector for the supervised machine learning model. We split the list of embeddings into 80% training and 20% testing data, resulting in 102,400 and 25,600 samples, respectively. The model was fit with the training data and the known compiler labels (i.e., which embedding mapped to which compiler, `clang` or `gcc`). This created our supervised model, ready for predicting the classification of future graph embeddings.

IV. Results

We performed our experimentation on a macOS desktop equipped with an Intel Xeon W-3235 with 24 virtual cores clocked at 3.3GHz with 96 GB of RAM running at 2933MHz. Analysis of the 2,972 executables to create an equal amount of `.bndb` files took 23 hours and 59 minutes, averaged 290.2% processor usage, and utilized a peak of 50.96 GiB of memory. Both the executables and the `.bndb` files are available at [44]. Constructing the 8,553,214 GraphML files from the executables, equal to the number of non-thunk functions uncovered by Binary Ninja, took 93 hours and 16 minutes, utilized an average of 162% of the processor, and a peak of 28.2 GiB of RAM.[6]

Since the Karate Club implementation of `graph2vec` does not have an iterative learning model, we had to load all of

[6]The use of multiple cores was only due to Binary Ninja opening `.bndb` files, and the remainder of the analysis was single threaded. In practice, this step of the analysis is highly parallelizable. To implement, limit the number of Binary Ninja workers to one thread.

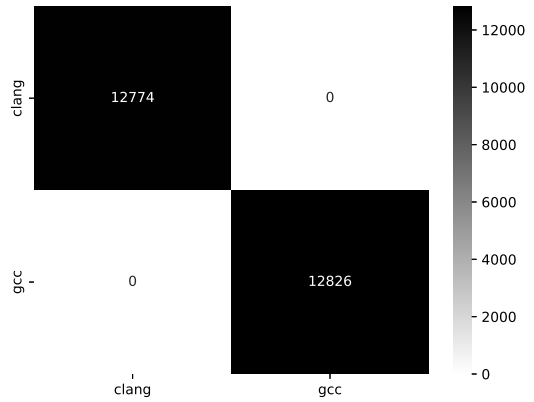


Fig. 2. Confusion matrix depicting the actual values versus the random forest's predicted values.

the graph embeddings into memory simultaneously to fit the model. Unfortunately, given our machine's constraints, we could not load all the GraphML files from the previous step into memory. Instead, we took a random sample of 64,000 GraphML files from both the `clang` and `gcc` graphs. We then fit `graph2vec` model with these NetworkX graphs. This step consumed 11.04 GiB of memory which took 24 minutes and 19 seconds at 100% processor utilization.

Fitting the Random Forest Classifier with the embeddings took 68 seconds at 100% CPU utilization - the quickest step in our process - and consumed 1.012 GiB RAM.[7] We then tested our model on the 20% testing data. With this, we found a *perfect* model that correctly predicted the results for each sample in the testing set. We present the results in the confusion matrix in Figure 2. Accordingly, we obtained a precision and recall score of 1.00. This result is commensurate with Pizzolotto and Inoue's findings in [38], where the authors achieved a 99.95% accuracy score when discriminating between `clang` and `gcc`.

We experimented several times with varying splits of the training and testing sets. We found that the model's accuracy did not drop below 99% until the testing set was 99.7% of the total samples, leaving just 384 graphs in the training set. With this small training set, the model achieved an accuracy score of 98.8%, an F1 score of 99.5%, and an area under the curve score of 98.1%. The excellent results with such limited training samples suggest the compilers frequently implement the same structural patterns across functions and that there is little benefit to training on additional samples compiled in the same manner.

V. Conclusion

In this paper, we described a workflow to automatically label executables, with no knowledge of the source code or information used to compile it, as being compiled with either `clang` or `gcc`. The immediate goal of our project was to determine if the code property graph constructed from the intermediate language lifted from the machine code of an executable would still yield meaningful information. Our results indicate that we can still determine structural differences in executables at the function level, even when those functions are lifted to higher level abstract representations.

We focused on determining compiler information because of the relative ease of obtaining a labeled dataset. Other experiments have demonstrated the ability to extract graphical structural differences using the code property graph to find vulnerabilities [15]. However, these experiments relied on representing the executable as a source code language, such as pseudo-C using Ghidra's decompiler [15]. Since we have demonstrated the implementation of the code property graph in an intermediate language and know that these languages can provide more semantic meaning [3] than assembly, we hope that this will be beneficial for automatically extracting additional information from executables, such as the likelihood of vulnerabilities appearing in a given function.

A. Future Work

We have several recommendations for future researchers who intend to implement our approach. Additional work is needed to improve the speed at which programs are decompiled into Binary Ninja database files and then saved in a GraphML format. Vector 35 released a performance release in version 3.1 in 2022 that improved analysis time by 300% when compared to previous releases [45]. It is unlikely for third-party developers or researchers to improve on those analysis times significantly. However, it is possible to perform analysis on separate executables in parallel using a tool like GNU Parallel [46]. Using multiple processes would increase the number of executables analyzed simultaneously.

Additional effort should focus on improving the construction speed of the code property graph. While the algorithm is linear with respect to the number of nodes in the abstract syntax tree, the current implementation makes additional passes over the tree to build the other graphs and to ensure accuracy.

This experiment considered only executables compiled for the `amd64` architecture to the `ELF` file format. However, given that BNIL is architecture agnostic, it is likely that the results of this experiment hold for other architectures, such as `mips` and `arm`, as well as other file formats like `Mach-O` and `a.out`.

We examined all of the functions that were not thunks in the executable. In [38], Pizzolotto and Katsuro found that they could infer compiler information with access to as little as 65 bytes of the program. Similarly, it may be possible to build the code property graph for slices of the program rather than explicitly creating graphs at the function level.

To further reduce the problem space, it may be the case that individual program functions, since they were compiled into a single program and not separately, share certain similarities that make training on multiple functions from the same program redundant. A future approach could construct the dataset

[7]Should this ever become a bottleneck, `sklearn` does offer an iterative learning method for the Random Forest classifier.

by looking solely at a single function from each program, such as the entry point. Fewer samples representing the same information would decrease the time spent at all stages of analysis.

BNIL implements SSA form to construct higher abstraction layers, like LLIL, MLIL, HLIL, and pseudo-C. SSA provides information on detected data dependencies but offers nothing about the conditions that led to the data creation. There is an additional representation, single static information (SSI) form, that would yield further information about the conditions that led to the creation of data [47]. Should an intermediate language natively expose SSI form, it may prove beneficial to embed that information into the edges of the program dependence graph for an additional training metric.

Finally, additional unsupervised graph neural networks exist that create embeddings that can yield higher classification accuracy, such as InfoGraph introduced by Sun, Hoffmann, Verma, and Tang in [48]. However, we did not observe these models implemented in Karate Club or any similar Python library. Pending implementation, future program classification tasks may benefit from these models.

ACKNOWLEDGMENT

The author thanks Ryan O'Neal and Austin Norby for assisting in interpreting results and providing mentorship during the project.

Binary Ninja™ and the Binary Ninja™ logo are registered trademarks of Vector 35. Express permission was given for their use to appear in this publication. IDA™ is a trademark of Hex-Rays. There was no intent to infringe on the rights of any trademark owner.

REFERENCES

[1] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *CoRR*, vol. abs/1803.04497, 2018. [Online]. Available: http://arxiv.org/abs/1803.04497

[2] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," 2021. [Online]. Available: https://arxiv.org/abs/2108.06363

[3] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 475–487. [Online]. Available: https://doi.org/10.1145/3395363.3397370

[4] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, no. 1, pp. 1–13, 2021.

[5] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of api calls," in *2010 Second Cybercrime and Trustworthy Computing Workshop*, 2010, pp. 52–59.

[6] J. Yuan and S. Ding, "A method for detecting buffer overflow vulnerabilities," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011, pp. 188–192.

[7] joernio, "Joern - the bug hunter's workbench." [Online]. Available: https://github.com/joernio/joern

[8] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "Cpgva: Code property graph based vulnerability analysis by deep learning," in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*, 2018, pp. 184–188.

[9] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 4665–4671. [Online]. Available: https://doi.org/10.24963/ijcai.2019/648

[10] Z. Haojie, L. Yujun, L. Yiwei, and Z. Nanxin, "Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network," in *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, 2021, pp. 250–255.

[11] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[12] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[13] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19. [Online]. Available: https://doi.org/10.1145/800028.808479

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[15] L. Zhou, M. Huang, Y. Li, Y. Nie, J. Li, and Y. Liu, "Grapheye: A novel solution for detecting vulnerable functions based on graph attention network," in *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, 2021, pp. 381–388.

[16] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," 2017. [Online]. Available: https://arxiv.org/abs/1707.05005

[17] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," 2016. [Online]. Available: https://arxiv.org/abs/1607.00653

[18] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1365–1374. [Online]. Available: https://doi.org/10.1145/2783258.2783417

[19] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan, "subgraph2vec: Learning distributed representations of rooted subgraphs from large graphs," 2016. [Online]. Available: https://arxiv.org/abs/1606.08928

[20] W. Mahoney and J. T. McDonald, "Enumerating x86-64–it's not as easy as counting."

[21] "Pcode operations." [Online]. Available: https://ghidra.re/ghidra_docs/api/constant-values.html

[22] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

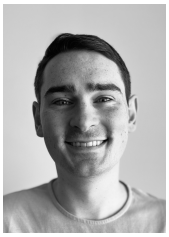
[23] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 2244–2253. [Online]. Available: https://proceedings.mlr.press/v139/cummins21a.html

[24] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, "Static analysis of energy consumption for llvm ir programs," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 12–21. [Online]. Available: https://doi.org/10.1145/2764967.2764974

[25] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo, "Skink: Static analysis of programs in llvm intermediate representation," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 380–384.

[26] E. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson, "Gtirb: Intermediate representation for binaries," 2019. [Online]. Available: https://arxiv.org/abs/1907.02859

[27] lifting bits, "McSema." [Online]. Available: https://github.com/lifting-bits/mcsema
[28] KyleMiles, "McNinja." [Online]. Available: https://github.com/KyleMiles/McNinja
[29] spinsel, "P-Code Reference Manual," September 2017. [Online]. Available: https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html
[30] I. Guilfanov, "Decompiler internals: microcode," August 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Guilfanov-Decompiler-Internals-Microcode-wp.pdf
[31] "Binary ninja intermediate language series, part 0: Overview." [Online]. Available: https://docs.binary.ninja/dev/bnil-overview.html
[32] BinaryAnalysisPlatform, "Bap." [Online]. Available: https://github.com/BinaryAnalysisPlatform/bap
[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.
[34] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 01 2009.
[35] G. Team. (2002) The graphml file format. [Online]. Available: http://graphml.graphdrawing.org/
[36] [Online]. Available: https://www.hex-rays.com/cgi-bin/quote.cgi/products
[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[38] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163 461–163 475, 2021.
[39] K. I. Davide Pizzolotto, "Binary software compiled for different architectures with different optimization levels," Apr. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4659370
[40] Vector-35, "binaryninja-api." [Online]. Available: https://github.com/Vector35/binaryninja-api
[41] B. Rozemberczki, O. Kiss, and R. Sarkar, "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM, 2020, p. 3125–3132.
[42] benedekrozemberczki, "Karate club," June 2022. [Online]. Available: https://github.com/benedekrozemberczki/karateclub
[43] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," 1 2008. [Online]. Available: https://www.osti.gov/biblio/960616
[44] S. Deaton, "Compiled executables and their resulting Binary Ninja database files," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6863087
[45] J. Wiens, "3.1 the performance released," May 2022. [Online]. Available: https://binary.ninja/2022/05/31/3.1-the-performance-release.html
[46] "Batch processing and other automation tips." [Online]. Available: https://docs.binary.ninja/dev/batch.html
[47] C. S. Ananian, "The static single information form," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.
[48] F.-Y. Sun, J. Hoffmann, V. Verma, and J. Tang, "Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization," 2019. [Online]. Available: https://arxiv.org/abs/1908.01000



Sean Deaton (Member, IEEE) is from Oʻahu, Hawaiʻi. He received a B.S. in computer science from the United States Military Academy at West Point, NY, in 2017 and an M.S. in computer science from the Georgia Institute of Technology in Atlanta, GA, in 2021.

From 2017 to early 2022, he was a vulnerability researcher with U.S. Army Cyber. Since 2022, he has been a senior vulnerability researcher with Blue Star Cyber. His research interests include binary exploitation, uncovering security flaws, machine learning, and capturing flags.