GNNIC: Finding Long-Lost Sibling Functions with Abstract Similarity

Qiushi Wu[1,2], Zhongshu Gu[2], Hani Jamjoom[2], Kangjie Lu[1]
{wu000273@umn.edu, Qiushi.Wu@ibm.com}, zgu@us.ibm.com, jamjoom@us.ibm.com, kjlu@umn.edu
[1] University of Minnesota, [2]IBM Research

***Abstract*—Generating accurate call graphs for large programs, particularly at the operating system (OS) level, poses a well-known challenge. This difficulty stems from the widespread use of indirect calls within large programs, wherein the computation of call targets is deferred until runtime to achieve program polymorphism. Consequently, compilers are unable to statically determine indirect call edges. Recent advancements have attempted to use type analysis to globally match indirect call targets in programs. However, these approaches still suffer from low precision when handling large target programs or generic types.**

This paper presents GNNIC, a Graph Neural Network (GNN) based Indirect Call analyzer. GNNIC employs a technique called *abstract-similarity search* to accurately identify indirect call targets in large programs. The approach is based on the observation that although indirect call targets exhibit intricate polymorphic behaviors, they share common abstract characteristics, such as function descriptions, data types, and invoked function calls. We consolidate such information into a representative abstraction graph (RAG) and employ GNNs to learn function embeddings. Abstract-similarity search relies on at least one anchor target to bootstrap. Therefore, we also propose a new program analysis technique to locally identify valid targets of each indirect call. Starting from anchor targets, GNNIC can expand the search scope to find more targets of indirect calls in the whole program. The implementation of GNNIC utilizes LLVM and GNN, and we evaluated it on multiple OS kernels. The results demonstrate that GNNIC outperforms state-of-the-art type-based techniques by reducing 86% to 93% of false target functions. Moreover, the abstract similarity and precise call graphs generated by GNNIC can enhance security applications by discovering new bugs, alleviating path-explosion issues, and improving the efficiency of static program analysis. The combination of static analysis and GNNIC resulted in finding 97 new bugs in Linux and FreeBSD kernels.

I. Introduction

Large programs, such as operating system (OS) kernels, often use indirect calls to achieve dynamic and polymorphic behaviors. For example, the Linux kernel 5.7 and Android kernel 5.10 have more than 55K and 62K indirect call sites. Accurately identifying these indirect calls is critical for building precise call graphs and control flow graphs, which are the basis of modern program analysis and security analysis, including bug detection [55], program debloating [1, 39], directed fuzzing [5, 35], vulnerability assessment [49], model checking [3], and many others [13, 37, 38, 47].

To identify indirect-call targets, recent works are primarily based on the following three methods: (1) Refining indirect call targets based on type analysis [28, 30]. Type analysis is widely used in indirect-call target identification because it is simple and can be sound in most cases. Specifically, it refines the indirect call targets by matching the type information of potential target functions and the function pointers. (2) Identifying and refining indirect call targets based on point-to analysis [24], which is a general approach to matching the pointer to its pointed addresses. (3) Refining the indirect call targets at runtime using control-flow integrity (CFI) [22] or other dynamic techniques [20].

However, the results of traditional approaches are still imprecise. According to our evaluation (see §VI-D), the precision rates for type-based approaches are typically less than 10%. This is mainly because type-based methods rely solely on type-related constraints for matching, without considering code semantics or behaviors. This issue is even more pronounced in large programs or when the type is too general. For instance, consider the function pointer `int (*console_blank_hook)(int)` in the Linux kernel V5.7, which has an `int` argument and an `int` return value. The type-based approaches match more than 1,200 target functions. However, in fact, there is only one real target function, `apm_console_blank` in the kernel. All other matches are false positives that merely share the same function type as this particular target. On the other hand, point-to-based approaches can suffer from both precision and recall issues. This is because precise pointer analysis is still an open problem for large programs. At last, indeed, dynamic analysis is precise, but its recall issue would make it less useful for analyzing OS-level large programs. According to the results of HFL [23] and DR.FUZZ [57], which are two recent kernel fuzzers, the code coverage is typically less than 10%, and even less than 1% for drivers.

Observations. Through an empirical study, we found that target functions of an indirect call typically share the same abstract behaviors (the most relevant behaviors to the major functionalities of a function), despite having different detailed behaviors due to polymorphism. We name the similarity across target functions as *abstract similarity*. For instance, in an operating system, there may be tens of file systems, each with its own implementation of the `write` function. The `write` functions are typically called through an indirect call based on the actual file system used. Although these `write` functions differ, they share the same abstract behavior — writing an

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23492
www.ndss-symposium.org

amount of data in a buffer to a file specified by a file descriptor. Therefore, when a known target exists for a specific indirect call, which is a practical assumption, as demonstrated later, abstract similarity can be used to identify other targets in the program.

One question that arises immediately is what kind of information can describe the abstract behaviors of a target function. Our empirical study shows that one direct source of information we can use is the representative information and the metadata of a function, which includes the function description, signature, and data types. In addition, we found that nested function calls also contribute to determining the abstract behaviors. We have further measured the significance of each kind of information in determining the abstract behaviors. In this project, we refer to such information that can represent the abstract behavior of functions as *abstractive information*.

Additionally, we found that graph structures are well-suited to represent abstractive information. This is because code semantics have been commonly represented in the form of graphs, such as call graphs and control-flow graphs. Furthermore, additional textual information, such as function names and descriptions, can be integrated into the graph nodes to capture and represent the human-understandable functionalities of functions. In the case of abstractive information, a graph structure can also accurately capture function invocations, data types, their relations, and textual descriptions.

In this paper, we propose using graphs to represent abstractive information and utilize Graph Neural Networks (GNNs) automatically learn and aggregate the abstractive information for deriving the abstract behaviors of functions. Our approach, named GNNIC, stands for Graph Neural Network (GNN) based Indirect Call analysis. By conducting the abstract-similarity analysis between the known target function of an indirect call and other functions, we can utilize GNNIC to identify more target functions for the indirect call.

Technical challenges. We anticipate two main challenges in implementing the GNNIC approach. First, we require at least one confirmed target function (referred to as an "anchor function") to start the matching process for more targets. Collecting anchor functions is challenging because static techniques often produce false positives, while dynamic-based methods have low coverage rates. Second, we need a reliable and versatile technique to represent abstractive information and build a precise similarity analysis for it. The abstractive information is diverse, including textual descriptions, data types, and function calls. We need a new representation technique to generally process such diverse information.

Key techniques. To address these challenges, we propose the following techniques. First, to identify anchor functions, we develop *scoped unique-name matching*. The technique is based on the fact that if a function pointer's name is unique in its dependency scope (i.e., possible scope in which the function pointer can be defined), any function assigned to the function pointer is most likely a valid target of an indirect call using the function pointer based on the same unique name. The technique thus focuses on generally identifying unique names and delimiting dependency scope. Second, we propose the use of a representative abstraction graph (RAG) that captures the diverse abstractive information of a function, including the descriptions and signatures of functions, the nested function calls, and data types. GNNIC trains a graph neural network against the RAG and generates embeddings for every function, based on which GNNIC can compare the abstract similarity between functions and anchor functions to match more targets.

We have built GNNIC upon LLVM, StellarGraph [9] and GraphSage [18]. We evaluated the effectiveness and performance of GNNIC with the commonly used large programs. We select OS-level programs, as they represent the largest and most complex programs. Experimented programs include the Linux kernel, Android kernel, and FreeBSD kernel. The evaluation results show that compared with state-of-the-art techniques, GNNIC can further improve the overall precision from 10% to 92.3%. Such an improvement is significant; for example, to identify a triggerable call chain in the Linux kernel through static analysis, on average, we can reduce the number of detected call chains from 10^9 to 10^4 based on the results of GNNIC (see §VII). Furthermore, GNNIC enhances traditional program analysis by reducing false positives and detecting previously missed new bugs.

We make the following research contributions in this paper.

- **Analyzing abstract similarity of functions.** We propose *abstract similarity* analysis as a general approach to refine indirect-call targets or assist other program analysis techniques. It performs well in cases (e.g., general types, large programs, unscalable pointer analysis) where existing approaches suffer from. It is complementary to existing approaches, such as type or pointer-based analysis, and can be used together with them.
- **Developing graph-based techniques for indirect call identification.** We propose *representative abstraction graph* (RAG), which is designed to capture diverse information (e.g., textual description, nested function calls, and data types) of functions and can be directly processed by GNN. Such an integrated and GNN-compatible representation may serve as a generic technique to enable many applications of GNN in program analysis. We also propose *scoped unique-name matching* as a precise technique to identify anchor functions for indirect calls.
- **Evaluating on a spectrum of security applications.** We present a comprehensive evaluation of the security applications with GNNIC. Our evaluation first showcases the effectiveness of GNNIC in improving the precision of bug identification. As a demonstration, we found 97 new NULL-pointer dereference bugs in well-established OS kernels, including the Linux kernel and FreeBSD kernel, by incorporating abstract similarity of functions with traditional static analysis techniques. In addition, we present multiple other security applications that can significantly benefit from the precise results generated by GNNIC. Overall, our comprehensive evaluation demonstrates the potential of GNNIC in improving the security of software systems.

II. Background and Study

Large programs usually use indirect calls to increase the flexibility and scalability of C and C++ code. In this study, we employ the illustrative example presented in Figure 1 to demonstrate the indirect calls. To complete an indirect call, the

process includes (1) taking the address of a function (target); (2) storing the address to a function pointer; (3) propagating the function pointer to an indirect call site; (4) dereferencing the function pointer and calling the target.

A. State-of-the-Art for Detecting Indirect-Call Targets

In theory, both point-to analysis and type analysis can be used to analyze indirect calls. However, due to the precision and scalability issues of the global point-to analysis, point-to-based approaches are not commonly used for identifying the indirect call targets in large programs. Therefore, the state-of-the-art works [26, 28–31, 46] use type-based approaches to handle indirect calls in programs. In this section, we discuss their limitations.

False positives resulted from generic types. Due to the following two reasons, applying the type-based analysis to large programs may trigger a large number of false positives. The first cause is related to generic types. When the types of function pointers are too general, such as `int (*console_blank_hook)(int)`, type analysis often performs poorly and falsely matches hundreds to thousands of targets. MLTA [28, 30] tries to alleviate this problem by involving more layers of type constraints, but it still struggles with generic types, often reporting thousands of targets for a single indirect call.

False positives resulted from global search. Another common problem with traditional type-based approaches is that they globally search for matched targets in the whole program. As type analysis does not track data flows, such a global search becomes a must, which, however, incurs many false positives. In a large program, many modules are not dependent on each other. For example, the Linux kernel has 15 well-defined subsystems, each containing numerous independent modules. Functions in modules that are independent of the module containing an indirect call can never become valid targets of the indirect call, even if their types match. Unfortunately, type matching is unable to delimit the search scope, which results in many false positives.

Out-of-the-scope cases of MLTA. To understand the cases that MLTA cannot handle, we have reused the two-layer type analysis implementation from [30] against the Linux kernel v5.7. The results show that many indirect calls cannot be handled by MLTA and must resort to single-layer type matching. There are two out-of-scope cases for MLTA. The most obvious case is that many function pointers do not involve a composite type (e.g., struct), i.e., they have a single layer of type, thus, cannot benefit from MLTA at all. Second, function pointers or object pointers are frequently cast to general-type pointers (e.g., `void *`), which also makes the function pointers disqualified for MLTA. Therefore, MLTA generates, on average, 84.7 potential targets for indirect calls in the Linux kernel, which is still very large, considering there are more than 55K indirect call sites in the Linux kernel.

B. An Empirical Study of Abstract Behaviors

The abstract behaviors of target functions. We refer to a *behavior* of a function as an operation against an object. Then, the abstract behaviors of a target function

```
1  /*****************Potential Target functions*****************/
2  s32 e1000_check_for_copper_link_ich8lan(struct e1000_hw *hw) {
3      struct e1000_mac_info *mac = &hw->mac;
4      s32 ret_val, tipg_reg = 0;
5      ...
6      ret_val = e1000e_phy_has_link_generic(hw, 1, 0, &link);
7      ...
8      e1000e_check_downshift(hw);
9      ...
10     ret_val = e1000e_config_fc_after_link_up(hw);
11     if (ret_val)
12         e_dbg("Error configuring flow control\n");
13     ...
14 }
15 s32 e1000e_check_for_copper_link(struct e1000_hw *hw) {
16     struct e1000_mac_info *mac = &hw->mac;
17     s32 ret_val;
18     ret_val = e1000e_phy_has_link_generic(hw, 1, 0, &link);
19     ...
20     e1000e_check_downshift(hw);
21     ...
22     ret_val = e1000e_config_fc_after_link_up(hw);
23     if (ret_val)
24         e_dbg("Error configuring flow control\n");
25     ...
26 }
27 /******************Function address taken*****************/
28 static const struct e1000_mac_operations ich8_mac_ops = {
29     .check_for_link     = e1000_check_for_copper_link_ich8lan,
30     ...
31 }
32 s32 e1000_init_mac_params_80003es2lan(struct e1000_hw *hw) {
33     mac->ops.check_for_link = e1000e_check_for_copper_link;
34 }
35 /*******************Indirect call site*******************/
36 static bool e1000e_has_link(...) {
37     ...
38     ret_val = hw->mac.ops.check_for_link(hw);
39     ...
40 }
```

Fig. 1: An indirect-call example in the Linux kernel.

are supposed to be the most relevant behaviors to the major functionalities of the function. In the example of indirect call `hw->mac.ops.check_for_link(hw)` (see line 38 in Figure 1), we can list and empirically rank the behaviors of each target function based on their relevance to its major functionalities. The most relevant behaviors of `e1000_check_for_copper_link_ich8lan()` include: checking the existence of the link, checking if there is `downshift`, configuring the link and checking configuration status, and checking and handling the link for different `mac` types; while the most relevant behaviors of `e1000e_check_for_copper_link` include: checking the existence of the link, checking if there is `downshift`, configuring the link, and checking the configuration status. By cross-checking the targets, we can clearly see that they indeed share three most relevant behaviors. In other words, these two targets share their abstract behaviors.

Furthermore, we have conducted a manual analysis of 501 target functions from 100 random indirect calls to verify if abstract behaviors are commonly shared between target functions of indirect calls. Our analysis showed that all indirect calls have targets that share at least one relevant behavior, with commonness decreasing as relevance decreases. This confirms that the most relevant behaviors, i.e., the abstract behaviors, are commonly shared across indirect call targets.

The abstractive information of target functions. Next, we aim to understand what information can be used to represent the abstract behaviors of a function. In pursuit of this goal,

we have examined 100 indirect calls and their corresponding target functions, as mentioned in the previous subsection for our empirical investigation. It is important to note that the findings presented in this section serve only as empirical or directional guidance. Additionally, our sampling methodology and its statistical validity are thoroughly addressed in Section §VI-A. Specifically, we first select possible behavior-related information, including the name of the target function, the data types used by the target function, the nested function calls, the control flow of the target function, and the data flow of the target function. We then empirically analyze which information is commonly shared across targets. The idea is that, as abstract behaviors are shared, we believe that commonly shared behavior-related information among target functions can be likely used to describe abstract behaviors.

The results show that the target functions for 97% of the indirect calls share similar names and descriptions and correspond to the behavior indicated by the function pointer name. For instance, as illustrated in Figure 1, the target functions of the indirect call exhibit a comparable function name, "check for link," signifying the analogous behavior of these functions. Additionally, we discover that the nested function calls and data types of target functions within an indirect call display similarity, boasting average Jaccard similarities of 0.62 and 0.41, respectively. This is determined by comparing the callee sets and utilized type sets of target functions. These values are markedly greater than the average similarity between randomly chosen functions, which are 0.33 and 0.23, respectively. Moreover, upon conducting a manual comparison of control and data flow among the target functions of a specific indirect call, we observe that only around 38% and 16% of them display similarity. To conclude, we can use function names and descriptions, internal function calls, and data types as the abstractive information of target functions.

Using GNN to represent nested abstractive information. Manually analyzing abstract behaviors is relatively straightforward for experienced programmers since people can understand and learn the descriptions, names, and nested callees of functions. However, automatically summarizing such abstract behaviors is not easy. This is because function calls and types are nested in programs. Suppose the tool cannot analyze such a nested relationship, in that case, it cannot accurately capture the abstractive similarity of functions. For example, `writeb` and `writew` further call `__raw_writeb` and `__raw_writew` respectively. The tool can accurately capture the similarity between `writeb` and `writew` only if it can understand the names of these functions and can further collect the abstractive information from their callees. Then, the tool can reflect all this abstractive information to the abstractive similarity between `writeb` and `writew`.

GNN is well-suited to catch such abstractive information of the nodes in the graph. The intuition of GNN is that given a node in the graph, GNN can aggregate the information of adjacent nodes to represent the given node. The neighboring nodes can further be described by their neighboring node. This feature is excellent for describing the nested abstractive information of functions in the call graph, because given a function in the call graph, we want to aggregate the abstractive information from its callee, which can further be used to describe the abstractive behavior of the given function. Similarly, these callees can be further described by the abstractive information of their callees. Therefore, GNN can accurately catch the abstractive behaviors of functions as we expect.

III. Overview

A. The Workflow of GNNIC

The goal of GNNIC is to identify indirect-call targets precisely, even for large programs. Figure 2 shows the overview workflow of GNNIC, which consists of four parts: ① collecting abstractive information, ② building representative abstraction graph (RAG) as GNN inputs, ③ collecting anchor functions for each indirect call, and ④ identifying more target functions using abstract similarity.

Specifically, by analyzing the source code and LLVM IRs of the target program, GNNIC first collects some basic information, including the representative information (e.g., function names, function descriptions, etc.), indirect call sites, all potential target functions, type-related information, the program call graph (without considering indirect call), etc. Then, GNNIC builds a RAG that integrates the call graph, representative information, and type-related information. It uses the RAG to train a graph neural network (GNN) model, based on which GNNIC can get a unique embedding for every function in the program. Furthermore, with a new technique, scoped unique-name matching, GNNIC identifies anchor functions for each indirect call in a delimited searching scope. At last, given an indirect call and its anchor functions, GNNIC uses the abstract similarity to identify more target functions.

B. Challenges and Techniques of GNNIC

Before showing the design of GNNIC, we first discuss the technical challenges (C) of realizing the GNNIC approach and the corresponding techniques (T) to address these challenges.

C-1: Collecting the anchor functions. To use abstract similarity to match more targets, GNNIC first requires anchor functions—at least one validated target function for an indirect call, which is still non-trivial. When talking about collecting real target functions, the most intuitive idea is using dynamic analysis because it would typically not introduce false targets. However, dynamic analysis against the system-level large programs suffers from a very low coverage rate. Thus, it is not a good solution for GNNIC. Additionally, preparing a fuzzing environment for a new program requires much effort and is time-consuming, such as generating quality seeds and preparing specific hardware for the driver. On the other hand, as we discussed, none of the current data-flow-based or type-based approaches can guarantee the precision of their indirect-call-targets identification. Therefore, new techniques are required for collecting anchor functions.

C-2: Digesting diverse abstractive information for similarity analysis. Given an indirect call, even if we have its anchor functions, it is still challenging to compute abstract similarity due to the diverse nature of abstractive information of target functions. Such information includes textual information, code semantics, and types. In addition, the relations among different kinds of information are also complex. Functions can call each other; complex types can contain each other; and functions can

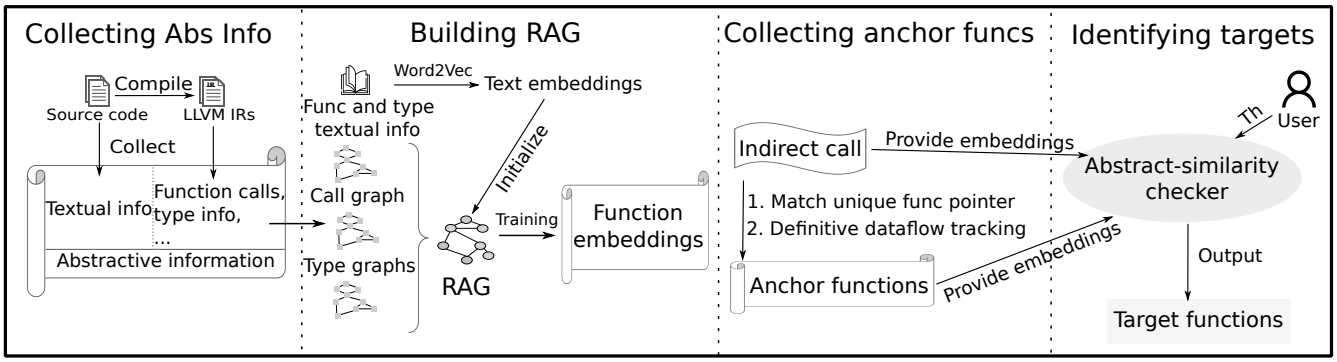

Fig. 2: Overview of GNNIC. RAG=representative abstraction graph, Abs Info=abstractive information, Th = threshold specified by user.

use different complex types. Thus, we need a systematic way to integrate diverse information, which should also be compatible with GNN.

T-1: Scoped unique-name matching for anchor functions. To address C-1, we propose a new technique to precisely identify anchor functions. In this step, we aim for high precision but allow for false negatives in identifying targets. Our technique is based on the fact that if a function pointer's name is unique in its dependency scope (possible scope in which the function pointer can be defined), any function assigned to the function pointer will likely be a valid target of an indirect call dereferencing the function pointer (with the same unique name). To realize the technique, we generalize the definition of "name" to support function pointers in structs and employ an iterative algorithm to narrow down the dependency scope for an indirect call. Our evaluation shows that the technique can identify at least one anchor function for 93.7% of indirect calls with a precision of 94%. We present the details of the technique in §IV-C. In addition, we find two scenarios where we can definitively determine anchor functions: callback functions and global variable related indirect calls. Thus, we also develop a definitive data flow tracking to precisely identify anchor functions for such indirect calls.

T-2: Catching diverse information with representative abstraction graph. To digest the diverse information about the abstract behaviors of a function, we propose the *representative abstraction graph* (RAG). RAG is not only capable of integrating various kinds of information, such as textual description, but, more importantly, it is also compatible with GNN. Its graph structure can be seamlessly processed by GNN. Specifically, RAG is essentially a graph structure in which each node is a function or type augmented with its representative information, and each edge represents the relation between nodes. By representing the abstract behaviors of functions with RAG, we are able to train a graph neural network for generating embeddings for abstract behaviors, which finally allows us to match more target functions from anchor functions based on the abstract similarity.

IV. The Design of GNNIC

A. Collecting Abstractive Information

As shown in our empirical study in §II-B, there are mainly three types of abstractive information that can reflect abstract behaviors of target functions: (1) function names and textual descriptions, (2) the function calls, as well as their relations, and (3) data types that are used by the functions. More specifically, the function names and descriptions are expected to be descriptive, which are intended to help programmers quickly understand the major functionalities of functions. Undoubtedly, they are important information that should be included for describing abstract behaviors. To a great extent, the nested calls of a function can also determine its abstract behaviors. This is because code reuse is a programming paradigm. In almost all programs, primitive functions (e.g., library functions for allocation, file operation, and memory management) are frequently reused. In other words, the parent function often acts like a synthesizer that logically organizes the primitive functions. Therefore, nested calls and their relations should also be included to describe the abstract behaviors of a function. Furthermore, the data types are the metadata pre-defining the objects that are operated by function behaviors, so they should also be included.

As an example, Figure 3 summarizes the abstractive information of several target functions for the indirect call, `hw->mac.ops.check_for_link(hw)` (also see line 38 in Figure 1). First, the function names and descriptions of these target functions are descriptive, indicating the abstract behavior that checks the status of some links in `e1000` module. Moreover, the shared callees of these target functions, such as `e1000e_config_fc_after_link_up` and `e1000e_phy_has_link_generic`, and the data types used by these target functions including `e1000_mac_info` and `e1000_hw` also indicate that the abstract behaviors of these target functions are `e1000` and "link" related. Therefore, all such information is included as abstractive information.

Collecting abstractive information of target functions. We can collect the data types and function call information through an analysis pass on LLVM IR and collect the textual information from the source code. Specifically, by analyzing IR, GNNIC first extracts the call relations between functions, the types used by functions, and the inclusion relations between types. Furthermore, GNNIC uses regex expressions to collect the descriptions of functions and types by scanning the source code.

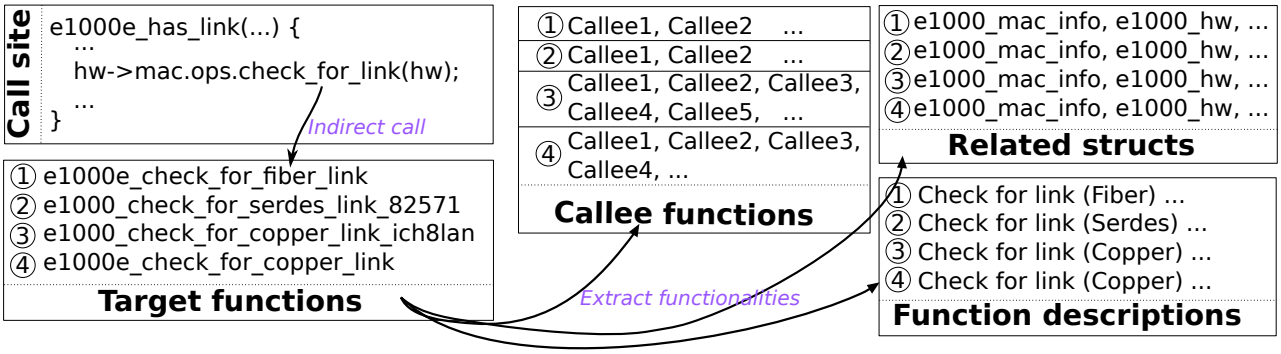

Fig. 3: Abstractive information of target functions. Callee1 to Callee5 are `e1000e_config_fc_after_link_up`, `e_dbg`, `e1000e_phy_has_link_generic`, `e1000e_check_downshift`, and `e1000e_get_speed_and_duplex_copper`.

B. Building RAG as GNN Inputs

As the next phase of GNNIC, it generates a vector for every function of the program, which can be used to evaluate the similarity of functions. Specifically, it includes three parts: (1) using vectors to represent the textual information, (2) generating representative abstraction graph (RAG) and, (3) leveraging the RAG to generate function embeddings.

Embedding textual information for the representation of functions and types. The names and descriptions associated with functions and types are considered as textual descriptions. Such descriptions are usually human-readable and encompass key information for understanding the functionalities. Thus, such information can assist GNNIC in evaluating the abstract similarity of functions. However, in order for programs to understand such textual information, we need natural language processing (NLP) techniques to turn such textual information into vectors. For example, by examining the description, the programmer can see that the "release" operation is similar to the "free" operation, which, however, is not quantifiable for programs. But after applying the NLP techniques, such two words can be changed into vectors with a short distance and thus can be quantified by other programs. Therefore, in this project, to easily integrate textual information into RAG and then be analyzed by GNN, GNNIC first equips NLP techniques to embed such textual information into vectors. Specifically, for each function and type, GNNIC separates its name into tokens and combines them with the descriptions (if any) as a single sentence. For example, for the function `e1000e_check_for_fiber_link`, GNNIC extracts "`e1000e`", "`check`", "`for`", "`fiber`", and "`link`" from its function name and also collects the comment "*Check for link (Fiber); Checks for link up on the hardware. If link is not up and we have a signal, then we need to force link up.*" from its function description [6]. Notice that we only consider the overarching descriptions of functions, excluding comments embedded within the functions' code. These internal annotations, while often elucidating specific variables or detailed operations, do not typically serve as summary descriptions of the whole function. GNNIC then pre-processes such text information based on the commonly used text pre-processing techniques [44], such as removing stop words and stemming. At last, by using Word2Vec [32, 33], the corpus of each function and type is embedded into a vector with 300 dimensions, which are used as the initial features for functions and types in RAG. To do so, GNNIC pre-trains a Word2Vec model based on more than 1.5GB textural corpus, which combines the code comments, documentation, git logs, and other program-related texts from multiple git repositories, including Linux, FreeBSD, OpenSSL, etc. In this way, the model can learn more descriptions and words related to programs.

Representative Abstraction Graph (RAG). Since there are three types of representative information, treating them separately can only yield one-sided results. Such information thus should be integrated at first. To this end, we propose RAG to leverage the graph structure reflecting the use, invocation, and containment relations between types and functions. Simultaneously, it incorporates individual node information, such as names and descriptions. To integrate such information, GNNIC first builds a separate graph for each relation between nodes (including nested functions and types) and then merges them into one graph, which is RAG. Finally, GNNIC initializes each node with its corresponding textual information.

Specifically, GNNIC first builds a directed graph to represent the function call graph (without considering indirect calls), indicating the function call relationship. Then, GNNIC builds a type usage graph that records all the functions and data types used by these functions. In this graph, the nodes are functions and types, and the directed edges point from functions to types indicating the usage relation. At last, GNNIC builds the type-relation graph, which describes the containment relations between different types. Each node is a specific data type, and the weighted directed edges point from the container types to its element types; the weights are the counts for the number of the specific element's types. For example, as a struct type, `struct e1000_mac_info` can be one of the nodes in the type-relationship graph, which includes one element with type `struct e1000_mac_operations`, three elements with type `u8`, twelve elements with type `bool`, etc. Therefore, in the type-relationship graph, from the node `struct e1000_mac_info`, there are at least three edges, an edge points to `struct e1000_mac_operations` with weight 1; an edge points to `u8` with weight 3; and an edge points to `bool` with weight 12. Based on all these relationship graphs and the embedding of textual information, GNNIC then merges

them into a RAG.

As shown in Figure 4, this merged graph is RAG, in which the nodes are functions and types with their embedding. Edges in the RAG can point from function to function, function to type, or type to type, which represents the relations between the caller with the callee, the type-user with the type, and the type with the types of its elements. After this, GNNIC trains a GNN against this RAG.

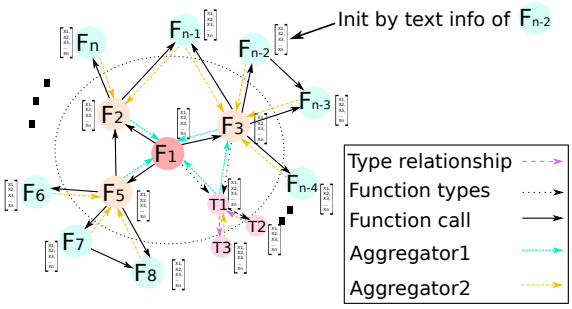

Fig. 4: Structure of GNN on representative abstraction graph. F=function, T=type.

Generating node embedding for RAG. This step aims to use an unsupervised approach to learn embeddings of functions and types only based on the RAG. To this end, we choose to use the state-of-the-art technique, GraphSage [18], to embed nodes in the RAG. We choose GNN mainly because it can effectively learn and utilize the relations between nodes in the RAG. In particular, when given a node in the graph, GNNIC aggregates the feature information of its neighbors and represents this node with the aggregated information. Given a node v in k-th layer, its embedding h_v^k can be expressed by the nodes in the (k-1)-th layer as follows:

$$h_v^k = \sigma(\mathbf{W} \cdot Mean(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in N(v)\})) \quad (1)$$

Here, $N(v)$ represents the sampled neighbors of node v; $\mathbf{W}$ indicates the trainable weight matrix; σ is a non-linear activation function such as ReLu. For example, as shown in Figure 4, the embedding of the function `F1` is generated by the aggregation of the abstractive features for its callee functions `F2`, `F3`, `F5`, and the contained type `T1`. Based on this approach, GNNIC can generate the embedding for every function in the program, representing the abstract behavior of functions.

C. Collecting Anchor Functions with Scoped Unique-Name Matching

As we discussed in §III, GNNIC should first collect anchor functions, with which GNNIC can further refine the indirect call targets by checking the abstract similarity between potential target functions and these anchor functions. The goal is to precisely identify at least one anchor function for as many indirect calls as possible. It is worth noting that this step never aims for completeness but for precision. Also, as we discussed in **C1** (see §III-B), due to the low coverage rate or high false-positive rate of traditional approaches, we need a new solution.

Intuitively, if a function pointer's name is unique in the program or in the scope of its dependencies, we can precisely match any assignment of a function's address to the function pointer with an indirect call using the function pointer based on the name, without the need for tracking the data flows. Any function whose address is assigned to a unique function pointer is most likely a valid target of an indirect call. Based on the insight, we propose *scoped unique-name matching* to identify anchor functions. Although in theory, it is possible that the function pointer can be re-defined, resulting in invalid matched targets, our evaluation in §VI-C shows that such a technique works extremely well for GNNIC—it can identify at least one anchor function for 93.7% of indirect calls with 6% false positives.

For example, in the Linux kernel, the name of function pointer `associate_indicator` is unique in the whole kernel. Also, we can find that the address of the function `lane2_assoc_ind` is assigned to this unique function pointer. Therefore, the function `lane2_assoc_ind` is an anchor function for any indirect call that uses this function pointer. Note that because of the uniqueness of the function pointer, we can know if an indirect call dereferences the function pointer. The power of such a unique name matching is that it eliminates the need for data flow tracking from function assignment to indirect call.

To make the technique general and precise, we still need to overcome two problems: (1) generalizing the definition of "name" for function pointers and (2) delimiting the dependency scope for an indirect call.

Defining unique names. In large programs, function pointers typically exist as fields of structs instead of as standalone pointers. Therefore, we define the name as a *composite name* that includes the name and type of struct objects, as well as the name of the function pointer. For example, for the function pointer on line 38 of Figure 1, its composite name is a combination of strings, `"e1000_mac_operations"`, `"ops"`, `"s32*(struct e1000_hw *)"`, and `"check_for_link"`, which contain the name and type of the function pointer, as well as its struct object. Such a representation of the name allows us to generally cover all kinds of function pointers.

Delimiting the dependency scope. Another problem we overcome is the scoping of dependencies. Apparently, indirect calls can only call the functions of modules that have data dependencies with the indirect call, as the addresses of the functions must be passed to the indirect call through data flows. Therefore, in this step, we delimit the scope to only data-dependent modules for the indirect call. We choose to perform the scoping at the granularity of the module to simplify the design and ease the implementation, which has worked reasonably well.

Specifically, GNNIC adopts a simple yet effective policy to define dependency. Two modules can have data dependencies only under the following two conditions, ① one module calls functions defined in another module, or ② one module uses the global variable defined in another module. Based on these two conditions, by iteratively considering modules with data dependency as the scope, GNNIC uses unique-name matching to identify anchor functions only within that scope. Notice that, to be precise, when finding the data dependency modules, GNNIC does not consider the data flow passed by indirect calls. Such a limited scope can significantly narrow down the

search space, so that the name of a function pointer has a high chance to be unique. For example, `ops.release()` is an indirect call in `e1000e` module. In the whole kernel, we can find hundreds of target functions whose addresses are assigned to the function pointers with the same name. However, after we delimit the scope to the modules that have data dependency with `e1000e`, only eight targets are identified and all of them are valid according to our manual analysis.

```
1  Type ST GV{
2      .func_pointer = foo1;
3  }
4
5  void CallerOfIcalls(Type (*FuncPointer)(Type A), ...) {
6      FuncPointer(A);
7      GV.func_pointer(...);
8  }
9
10 void CallerOfIcaller(...) {
11     CallerOfIcalls(foo2,...);
12 }
```

Fig. 5: A simplified example for definitive data flow tracking.

Definitive data flow tracking for function pointers. In addition to using scope unique-name matching, we find two scenarios in which standard data flow tracking suffices to identify anchor functions: (1) call back as a function argument, and (2) direct use of a function pointer in a global initializer.

Specifically, for the call-back functions, the function address is typically directly stored to a function pointer as an argument, which is typically used in the current function. For example, in Figure 5, on line 11, the function address of `foo2` is directly passed to `CallerOfIcalls` and used in it. Through an intra-procedural backward data flow analysis, GNNIC can precisely know that `foo2` is an anchor function.

On the other hand, a global object is often initialized through a *global initializer* that clearly stores the address of a function to a function pointer in the object. When an indirect call uses the global, we can precisely know the function is an anchor function. For example, in Figure 5, `func_pointer` in the global variable `GV` is initialized as `foo1` (on line 2). Thus, `foo1` is an anchor function for the indirect call on line 7. Our evaluation shows that GNNIC can use such definitive data flow tracking to find anchors for 2% of indirect calls.

D. Identifying More Targets with Abstract Similarity

After we get the embedding of each function and the anchor functions of each indirect call, GNNIC then tries to identify more target functions using abstract similarity.

Similarity analysis. Since we have the embedding of abstract behaviors for both anchor functions and other functions, the idea is to compute the similarity of embeddings. Given an anchor function (or sometimes multiple anchor functions) and a to-be-matched candidate function, GNNIC evaluates the similarity based on the following expression.

$$S_c = MEAN(\{\frac{\mathbf{v}_a \cdot \mathbf{v}_c}{\|\mathbf{v}_a\|\|\mathbf{v}_c\|}, \forall \mathbf{v}_a \in G(\mathbf{v})\}) \quad (2)$$

Here, let us assume `func_t` is a candidate target function. Then, $\mathbf{v}_c$ is the embedding for the `func_t`; S_c represents the similarity between this function and the anchor functions; $\mathbf{v}_a$ is the embedding for an anchor function; $G(\mathbf{v})$ represents the set for all anchor functions of the indirect call. GNNIC uses the average number for the similarity between `func_t` and all anchor functions of the indirect call, which can show the abstract similarity of `func_t` with the ones of anchor functions. The similarity, S_c, always belongs to the interval `[-1,1]`; the larger S_c is, the more similar `func_t` and the ground truth are. Therefore, $S_c = 1$ means `func_t` has the same functionality with at least one anchor function. Thus `func_t` very likely belongs to the real target function of the indirect call. Oppositely, $S_c = -1$ means `func_t` is totally different from all the anchor functions. Thus `func_t` unlikely belongs to the real target function of the indirect call.

This approach makes the inputs and outputs of GNNIC flexible. First, GNNIC can refine the results of different traditional indirect-call analysis techniques such as type analysis and multi-layer type analysis. Second, GNNIC can easily adjust the false-positive rate and false-negative rate of the results by changing the threshold (S_c) to fit the precision-driven applications or the recall-driven applications. Third, our evaluation results reveal a notable similarity among anchor functions involved in indirect calls with multiple anchors, as evidenced by a medium similarity of 0.79 and an average of 0.78. Such a feature enables incorporating multiple anchor functions to ensure system robustness with minimal false positives, providing reliable and accurate results. Note that GNNIC is complementary to existing approaches. For example, it can be used to refine the results of type analysis by further removing functions reported by type analysis that are not similar to anchor functions in abstract behaviors.

V. Implementation of GNNIC

We have implemented GNNIC based on LLVM and GraphSage [18]. This section presents several implementation details of GNNIC.

Extracting the name of function pointers. As discussed in §IV-C, in order to collect anchor target functions, GNNIC collects and matches the unique function pointers in a limited scope. Collecting the name of the function pointer and container struct at the indirect call site is done by a Python script code, including mainly two parts, collecting function-pointer initializers and expanded Macros. Here we did not directly work on the pre-compiled code, in which the Macros are expanded. This is mainly because we can only map the instructions in the LLVM IR into the source code lines based on the debug information. However, we cannot directly map such instructions to the pre-compiled code. Specifically, given an indirect call instruction, GNNIC first collects its source code line based on the debug information provided by LLVM. Further, GNNIC checks if a Macro does have such an indirect call. If the answer is yes, GNNIC further looks for the function-pointer name and struct name inside the Macro. Otherwise, GNNIC directly extracts such names. This approach works well for most indirect calls; however, it cannot correctly handle some corner cases, such as the function pointer name being formed by multiple strings pasted by token-pasting operators.

Handling no anchor function cases. Based on our evaluation (see §VI-C), GNNIC can detect at least one anchor function

for 93.7% of indirect calls. For the uncovered 6.3% of indirect calls, we choose to "borrow" the anchor functions from similar indirect calls to represent their abstract behaviors. Specifically, given an uncovered indirect call, GNNIC first finds another indirect call that is most similar to it by comparing the name and type of the function pointer. Then, GNNIC regards the anchor functions of that similar indirect call as the anchor functions for the uncovered indirect calls. This decision is based on the results of the background study, which show that in most cases, the name of function pointers is highly related to the abstract behavior of the indirect call. Therefore, similar indirect calls also tend to have similar function pointers and abstract behaviors. This is essentially another application of abstract similarity, where it is applied to indirect calls instead of functions. More evaluation related to the anchor functions can be found in §VI-C.

Handling RAG with indirect callees. When building the RAG, GNNIC incorporates the call graph, which is part of the abstractive information we include. A target function may have many nested calls, and some of them are indirect calls. When a nested call is an indirect call, the nested callee is missing and cannot be incorporated into the abstractive information of the target function. To handle this problem, we borrow the identified anchor functions of indirect calls to complete the missing nested callees. Such a strategy can slightly improve the overall stability of the model when we use RAG to train the function embeddings.

Setting up the GNN model. We use the directed GraphSage [18] model, which is implemented by the library StellarGraph [9], to handle RAG. Specifically, we use two layers in the GraphSage encoder, with a 10 and 5 sample size for the first and the second hops. We run five epochs with batch size 10K. The input of the mode is RAG, the nodes of which are initialized by the vector representing the textual information with 300 dimensions.

VI. Evaluation

A. Experiment Setting and Data Sets

Platform. We use two computing resources available to us, including a server (8 cores/60GB memory/a single GPU) and a desktop (24 cores/64GB memory/a single GPU). Both of these two machines are running on Ubuntu 20.04.

Ground-truth targets for false-negative evaluation. To evaluate and compare the false-negative of GNNIC under different settings, we must first have a set of ground-truth targets. To this end, we downloaded and analyzed the previous fuzzing logs from the results of Syzbot [17], from which, we identified 3,831 unique indirect caller-callee pairs. To acquire these function pairs, we first extracted all caller-callee pairs in the fuzzing logs reported by Syzbot, whether or not the corresponding cases trigger bugs. Because all of them are resources for supplying us with valid caller-callee pairs. Consequently, we have collected a collection of 11,286 fuzzing logs, all of which were reported before the year 2022. Furthermore, we collected the indirect caller-callee pairs from them by leveraging MLTA, which has no false negatives. Any overlooked pair in this dataset would signify a false negative of GNNIC. Such results can be used to evaluate the false negatives of GNNIC conservatively. Notice that the real false-negative rate of GNNIC must be lower than the estimation provided by this method, due to the false positives of the MLTA. For instance, if a caller directly invokes a callee while a function pointer is dereferenced in this caller function, the MLTA might incorrectly identify the same callee as an indirect call target, when the function pointer and the callee share the same types. Our current method would classify such misidentifications as ground truth, increasing the estimated false negatives of GNNIC. Although these cases are relatively rare, as evidenced by our manual analysis.

We adopt this approach for several reasons. First, it allows us to gather sufficient data points, enhancing the statistical confidence of our analysis. Second, it helps minimize human bias in the evaluation of false negatives because humans often focus on confirming simpler true positives based on involved functions and pointer names while overlooking complex data flows. Third, we leverage open fuzzing logs, which offer comprehensive coverage, instead of relying solely on self-executed dynamic analysis. This approach overcomes limitations such as exploring a limited number of paths within a constrained time frame and the inability to access specific devices where certain execution results may be unattainable.

Method for false-positive evaluation. Most of the state-of-the-art works(e.g., ICallee [59], TypeDive [28], Crix [30]) only use the refining rate of indirect-call targets to show the effectiveness of their tools compared with the previous approaches. However, none of them has evaluated the real false-positive rate for the indirect-call target identification, mainly due to the program complexity, which makes such evaluation time-consuming. To fill this gap and demonstrate the concrete performance of the current state-of-the-art, we manually performed a false-positive analysis. Specifically, we first sampled 100 indirect caller-callee pairs from the results of each following method: our work, type analysis, and type+multi-layer type analysis (MLTA). Then, we manually analyze such pairs to see if the target function can be a real target function or a false positive based on the following methods. ① If we can find at least one path that can pass the address of the target function to the call site, then we believe it is a true target function. ② If we cannot find such paths and the functionality of the target function is obviously different from the intention of the function pointer Then we believe it is a false target. Otherwise, we will consult as many auxiliary materials as possible, such as a fuzzing log, to determine whether the function is the real target function.

Data sampling for manual analysis. Static analysis techniques often rely on manual inspection of code segments or results to design or evaluate systems. Given the time-intensive and complex nature of the cases, manual analysis usually focuses on a sample of cases. When the total number of cases is substantial, state-of-the-art practices [4, 12, 15, 30, 43, 48, 49, 55] typically sample and manually analyze 40-400 cases. According to the previous study [7], such sampling ratios would result in a margin of error between ±5% to ±15%. In line with these state-of-the-art works and considering the complexity of manual analysis, we choose to select 100 - 300 samples for our case study (see §II-B) and false positive analysis respectively, ensuring that the margin of error falls within ±5% to ±10%, thereby providing statistically valid results. And to make sure the randomness of our sampled cases, we use pseudo-random number generators [11] to choose the data from the whole data

set.

B. Scalability of GNNIC

GNNIC can analyze system-level programs in hours. For the Linux kernel and the Android kernel, which has more than 20 million lines of code, it takes about 10 minutes to collect all abstractive information from their corresponding LLVM IR. Furthermore, it takes about three hours to pre-train Word2Vec, build RAG, and train GNN with RAG. Notice that people do not need to re-train the Word2Vec model when analyzing different target programs because they are reusable. Also, benefiting from the inductive learning of GraphSage, people do not need to re-train the whole GNN model after some new functions are added to the target program. At last, it takes about 10 minutes to generate the anchor functions and the final indirect call results. Thus, in total, it will take less than 4 hours to analyze the Linux or Android kernel. And the analysis time can be reduced to less than one hour for analyzing the FreeBSD kernel.

C. Evaluation on Anchor Function Identification

In this section, we assess the effectiveness of the anchor function identification process by examining instances of both false positives and false negatives.

The precision of the anchor functions. As discussed in §VI-A, we have sampled and evaluated the precision of the 100 anchor functions. Specifically, in the 100 sampled indirect caller and target pairs, 94 of them are valid targets, and 6 are false positives. This result indicates that most of the anchor functions are valid targets. Moreover, after looking into the 6 cases, we find that all of them are caused by the implementation issue of name matching. Because on the source code level, GNNIC cannot perfectly catch the name of the function pointer and its container struct when meeting some complex code, such as nested complex macros. Such issues would require extensive engineering effort to resolve; however, they would not significantly enhance the overall system. Thus, we propose to regard them as potential future work.

Failures in anchor identification. Besides the precision, we also evaluate the failures in anchor identification where GNNIC cannot find any anchor functions for an indirect call. Specifically, for the Linux kernel, GNNIC fails to identify anchor functions for 6.3% of indirect calls. By manually looking into 50 failure cases, we found the following causes. ① 56% of them are caused by the complexity of source code, such as using complex Macros to initialize function pointers. ② 44% of these issues stem from function pointers that never directly take the target functions' address but instead acquire it from other function pointers. In §VIII, we will further discuss covering more anchor functions in the future.

D. The Precision Improvements on Target Identification

We compared GNNIC with the two-layer type analysis, which represents MLTA [28, 30], for the following reasons. First, to ensure a fair comparison of GNNIC, it is essential to evaluate it against source-code level or IR-level program analysis techniques, rather than binary-based approaches. Notably, recent methodologies, such as iCallee [59], specifically focus on binary code, which inherently lacks certain information when compared to IR-level analysis. Consequently, comparing GNNIC to such approaches will introduce biases that may potentially underestimate the contribution of their results. Second, based on the results of TypeDive [28], other IR or source-code level methods, such as pointer analysis, are not scalable for system-level programs and typically do not even surpass MLTA [28]. Third, using GNNIC with MLTA is scalable and can minimize false negatives, benefiting downstream applications.

In this evaluation, we tested the performance of GNNIC on the Linux kernel. As type analysis globally matches targets and finds a superset, we apply GNNIC on the results of type analysis to refine the target functions. Notice that, as we discussed in §II, not all the indirect calls qualify MLTA because some indirect calls only have one-layer type information. In this scenario, we step back to use the results of the one-layer type analysis.

The reduction rate of target functions. Figure 6 illustrates the proportion of target functions that can be reduced by the GNNIC system under varying threshold settings. In this graph, the x-axis represents the abstract similarity (the threshold) between the candidate target function and the anchor functions, as described in Section IV-D. The y-axis corresponds to the reduction rate of target functions at a specific threshold. As the abstract similarity threshold rises, the GNNIC system is capable of filtering a greater number of target functions, the majority of which are considered invalid. For example, for the Linux kernel, GNNIC can reduce about 88% of target functions when the abstract similarity is larger than 0.9.

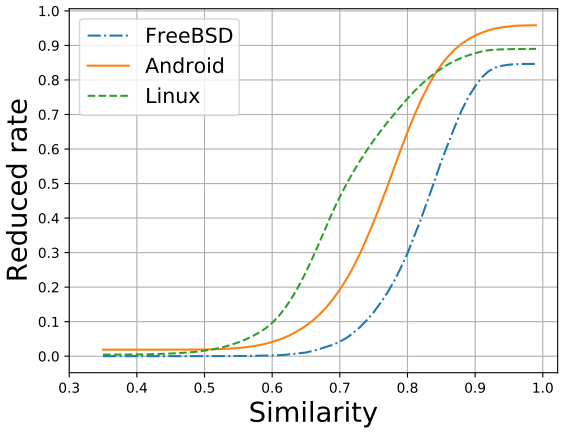

Fig. 6: Percentage of refined indirect-call targets for different OS kernels.

Receiver operating characteristic (ROC) of GNNIC. To more effectively evaluate the performance of the GNNIC system, we examine the relationship between the false-positive rate (FPR = FP/(FP+TN)) and true-positive rate (i.e., recall, TPR = TP/TP+FN), reflecting the false negatives of GNNIC, in Figure 7. We used the 3,831 ground-truth function pairs collected from fuzzing logs (see §VI-A) to assess the true-positive rate evaluation of GNNIC. Furthermore, as mentioned in Section VI-A, we sampled and manually analyzed 100 indirect caller and callee pairs from the MLTA results. Our aim

is to evaluate the false-positive rate of GNNIC based on these samples. To further minimize random error and to ensure the smoothness of the ROC curve, we further randomly sampled extra 200 pairs, bringing the total to 300 indirect caller and callee pairs from the MLTA to assist the false positive evaluation. Figure 7 demonstrates the effectiveness of GNNIC in achieving a desirable balance between false positive and true positive rates. At a false-positive rate of 0.33%, GNNIC achieves a true-positive rate (recall) of 84.8%, indicating 15.2% of false negatives. Under this condition, all the identified target functions are anchor functions. Conversely, when the true-positive rate (recall) reaches 99.6%, the corresponding false-positive rate rises to 60.7%. This underlines the precision of the GNNIC system in identifying the target functions of indirect calls.

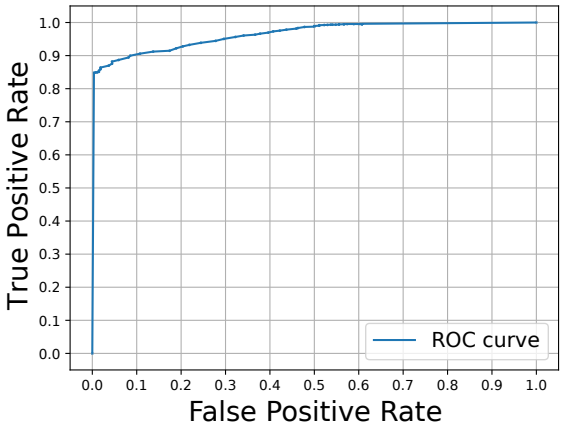

Fig. 7: ROC curve for results for GNNIC on the Linux kernel.

Comparing with type-based approaches. A comparison between the results of GNNIC and type-based approaches against FPR can be problematic, as determining the total number of negative targets within the entire program for type analysis is challenging. This difficulty makes it hard to compute the false-positive rate. Nonetheless, we can still draw a comparison between the precision (precision = TP/(TP+FP)) and recall of GNNIC and type-based approaches. To this end, by manually checking all these 300 indirect caller and callee pairs, we can find 30 of these cases are true positives, which indicates that the precision of MLTA is 10%. Based on the mathematical proof of MLTA [28], in theory, there should be no false negatives, and thus the recall rate of MLTA is 100%. For GNNIC, when its precision reaches the highest value of 92.3%, the corresponding recall value is 84.8%. This precision aligns with the 94% precision of anchor functions, as discussed in Section VI-C. These findings suggest that GNNIC can improve precision by up to 82.3% compared to MLTA, while missing 16.2% of real targets.

E. Evaluating GNNIC *on Different Projects*

Besides the Linux kernel, we also evaluated GNNIC over the Android kernel and the FreeBSD kernel.

Distribution for the number of targets. We also evaluated the distribution for the number of indirect-call targets identified by the type-based approach and by GNNIC. Similar to the previous implementations, we reused the implementation of a type-based approach [30]. Table I shows the distribution for the number of identified indirect call targets in the Linux kernel, Android kernel, and FreeBSD kernel. The percentage in the table indicates the percentage of indirect calls that have the corresponding number of target functions. The "Mean" and "Max" indicates the average and the maximum number of target functions for a specific indirect call. The results in Table I show that for 7.2%, 7.2%, and 2.8% of indirect calls in the Linux kernel, Android kernel, and FreeBSD kernel, the type-based approach will detect more than 100 target functions for them. However, GNNIC can reduce the corresponding number to 1.3%, 0.6%, and 0.2%, which shows that GNNIC can effectively refine the indirect call targets identified by the traditional type-based approaches.

Improvement for different kernels. Furthermore, Figure 6 also shows the percentage of reduced target functions based on the different similarity thresholds. Specifically, GNNIC can reduce 86% and 93% of target functions for the FreeBSD kernel and the Android kernel, which means GNNIC can effectively reduce the irrelevant target functions for different programs. However, due to missing enough ground truths for the FreeBSD and the Android kernel, we cannot draw the recall-precision curve for them.

VII. Security Applications

A. Leveraging Abstract Similarity for Enhanced Bug Detection

We evaluate the effectiveness of GNNIC in real-world security applications by integrating it with static analysis tools for finding security bugs in the Linux and FreeBSD kernels. Our analysis uncovered a significant number of bugs, which we are currently reporting to the developers. To prioritize security, we will withhold the disclosure of these bugs until we have ensured they are thoroughly resolved.

Enhancing bug detection in similar functions through refined cross-checking techniques. Cross-checking techniques are widely utilized in static analysis for bug identification, as they collect similar code snippets and consider deviations as potential bugs [30, 54, 56]. However, these techniques face challenges, including requiring a large number of code snippets to establish correct usage patterns, cannot handle infrequent cases, and hard to identify deviations in certain situations. To address these limitations, we propose a combination approach to improve bug detection. We aim to identify overlooked bugs for uncommonly used functions, by combining our technique with cross-checking. We first cluster functions with abstract similarities and apply cross-checking to the entire class of functions, allowing for analysis of functions not commonly used and tolerating more noise. To illustrate its effectiveness, we use "allocation" as an example of abstract behavior and manually selected misused allocation functions as anchor functions, such as `mlx4_alloc_cmd_mailbox`, from APISan [54] and Crix [30] results. Table II shows that our analysis of the Linux and FreeBSD kernels found 97 missing return value check bugs for 38 different allocation functions, with 63% (24/38) of those functions used less than 10 times in the program. All of these instances have the potential to trigger NULL-pointer dereference problems when the unchecked return pointers are further dereferenced. Traditional cross-checking techniques in Crix and APISan failed to report these bugs due

TABLE I: Distribution of indirect calls that have a number of targets in the range (specified on the first row). Type-based: Original type-based approach on the system; Sim=0.9: The results of GNNIC based on the similarity at 0.9.

System/ # of targets	<10	10-100	100-1000	>= 1000	TotalTargets	Total Icalls	Mean	MAX
Linux (Type-based)	77.0%	15.8%	5.4%	1.8%	4734762	55921	84.7	8862
Linux (GNNIC, Sim = 0.9)	85.7%	13.0%	1.3%	0.0%	545452	55921	9.7	2436
Android (Type-based)	82.8%	10.0%	5.6%	1.6%	4769716	62618	76.1	9056
Android (GNNIC, Sim = 0.9)	94.9%	4.5%	0.6%	0.0%	297284	62618	4.7	2645
FreeBSD (Type-based)	85.5%	11.7%	1.3%	1.5%	251307	7578	33.2	1960
FreeBSD (GNNIC, Sim = 0.9)	88.5%	11.3%	0.2%	0.0%	34669	7578	4.5	217

to their limitations in analyzing infrequently used or misused functions with a higher deviation proportion than the threshold. Such results demonstrate that combining our method with cross-checking can effectively identify more code issues, by reducing the false negatives of the original cross-checking-based techniques. On the other hand, GNNIC can also help reduce the false positives of traditional static analysis by providing them with more precise call graphs. A more in-depth discourse on this aspect will be presented later.

TABLE II: Misused "allocation" related functions in the Linux and FreeBSD kernels. ITT = identified total used times; IMT = identified misused times (bugs); F = flag to indicate if the cross-checking-only approaches can find the issue; Linux kernel is at git commit dbd736c8116f; FreeBSD kernel is at git commit 4ba619efbdd2476;

Target	Misused function name	ITT	IMT	F
Linux	proc_net_mkdir	7	1	Y
Linux	nci_skb_alloc	9	1	Y
Linux	bio_alloc_bioset	5	1	Y
Linux	kcalloc_node	14	1	Y
Linux	dma_alloc_attrs	19	1	Y
Linux	alloc_wrb_handle	4	1	Y
Linux	cdns3_gadget_ep_alloc_request	2	1	N
Linux	v4l2_ctrl_new_custom	23	4	Y
Linux	drm_atomic_get_new_connector_state	6	1	Y
Linux	acpi_os_allocate	5	2	Y
Linux	acpi_os_allocate_zeroed	7	1	Y
Linux	acpi_os_allocate_zeroed	3	1	Y
Linux	nfp_cpp_mutex_alloc	3	1	Y
Linux	alloc_irq_cpu_rmap	9	1	Y
Linux	pblk_alloc_rqd	4	2	N
Linux	__alloc_object	2	1	N
Linux	alloc_buffer_head	2	1	N
Linux	alloc_page_buffers	3	1	Y
Linux	kcalloc_node	14	1	Y
Linux	nci_skb_alloc	9	1	Y
Linux	alloc_pages_node	12	3	Y
Linux	v4l2_ctrl_new_int_menu	17	1	Y
Linux	i2c_new_device	9	1	Y
Linux	v4l2_ctrl_new_std_menu	32	2	Y
Linux	compat_alloc_user_space	15	15	N
Linux	iova_magazine_alloc	3	3	N
Linux	nd_dax_alloc	2	2	N
Linux	usbhs_pipe_malloc	2	2	N
FreeBSD	xpt_alloc_ccb	8	3	Y
FreeBSD	cam_simq_alloc	27	1	Y
FreeBSD	nvme_allocate_request_vaddr	10	5	N
FreeBSD	g_malloc	31	13	Y
FreeBSD	devfs_alloc	3	1	Y
FreeBSD	if_alloc	69	3	Y
FreeBSD	bit_alloc	2	1	N
FreeBSD	sglist_alloc	12	1	Y
FreeBSD	buf_ring_alloc	4	1	Y
FreeBSD	uma_zalloc	64	14	Y

Interesting findings during the bug detection. Incorporating abstract similarity of functions with static analysis not only enhances the capabilities of cross-checking techniques but also reveals new bugs that conventional cross-checking methods would completely miss. This is because if a function is used in a single, uniform pattern, the cross-checking technique cannot detect any issues. For instance, if the return value of a function is never checked throughout the entire program, the cross-checking approaches, such as Crix, will not identify the missing check issue. Similarly, if most of the usages (>=50%) of a function are incorrect, cross-checking methods would also miss them. However, such bugs can be identified by integrating abstract-similarity-based clustering. Due to such reasons, 33 out of 97 missing NULL-check bugs against 10 allocation functions would be totally missed by cross-checking-only approaches focusing on security checks. More interestingly, the return values from 3 of these allocation functions were never checked by security checks and thus will have gone unnoticed using only cross-checking techniques. Therefore, combining abstract similarity and static program analysis dramatically increases the ability to find new and previously overlooked bugs in code.

Reducing false positives in static analysis. Static analysis tools are widely used in both industry and academia to enhance software security and quality. Indirect-call analysis is critical for these tools, but false positives can impede bug detection. For example, as the claims from INCRELUX [55] and DiffCVSS [49], using traditional techniques, such as type analysis, to identify indirect calls will introduce high false positives into their results and sometimes even make the tool unusable. Our study found that imprecise indirect call analysis is a major contributor to the results of Crix, accounting for 59% of the total warnings. Integrating our system with Crix reduced these cases by 74%, resulting in a 44% reduction in total false positives. This demonstrates the potential of our system to deliver high-accuracy results when combined with conventional program analysis techniques.

B. Other Potential Security-related Applications of GNNIC

Enhancing vulnerability-reachability analysis. Another potential security application of the GNNIC is assessing the reachability of a vulnerability [53]. An analysis of call chains from Syzbot [17] reveals that, on average, there are 4.7 indirect calls in a chain within the Linux kernel. Also, based on the average number of target functions identified by GNNIC and type-based approach (see §VI-E), we can infer that in order to find the correct call chain; we need to analyze $84.7^{4.7}$ (10^9) candidate call chains based on the call graph constructed by the traditional approaches. However, with the accurate call graph built by GNNIC, we only need to analyze $9.7^{4.7}$ (10^4)

candidate call chains. This result indicates that the call graph built with GNNIC can dramatically improve the precision of call chain identification, thereby supporting future security-related research, such as vulnerability assessment.

Expanding bug identification capabilities with abstract similarity. In §VII-A, we demonstrated the potential of abstract similarity in enhancing bug identification. Our example of missing NULL checks for allocation-related functions showcases the effectiveness of our approach. However, it is essential to recognize that the application of abstract similarity is not confined to this specific type of bug and can be employed to detect other common bugs as well. Future research could concentrate on investigating the use of abstract similarity for identifying various bug types, such as permission issues in critical APIs. As highlighted in PEX [56], abstract similarity holds the potential for aiding in the detection of similar permission issues. Although this kind of research is time-consuming and beyond the primary scope of this paper, it offers exciting prospects for further exploration.

Improving directed fuzzing and concolic execution. Directed fuzzing and concolic execution often require knowledge of indirect-call targets during the execution. However, due to accuracy limitations or performance issues or existing approaches, only a few existing tools effectively handle indirect calls, such as HFL [23] and CollAFL [13]. Conversely, many dynamic analysis tools and concolic-execution tools, such as previous works [8, 19, 21, 25, 36–38, 40, 47, 58], do not handle indirect branches at all. However, neglecting to correctly handle indirect calls result in dynamic analysis tools missing many targets. For example, Böhme et al. [5] proposed a Greybox Fuzzing (DGF) based on the AFLGo, which found that *"more than half of the changed basic blocks are accessible only via register indirect calls or jumps (e.g., from function-pointers). Those do not appear as edges in the analyzed call-graph or in the control flow graph."* These findings suggest that over-approximated approaches, like type-based analysis, may generate numerous false positives, affecting downstream application accuracy and increasing analysis time. On the other hand, under-approximated approaches may introduce false negatives in downstream applications. Consequently, an accurate call-graph construction tool like GNNIC holds significant potential for enhancing existing directed fuzzing and concolic execution instruments. Furthermore, users can optimize the balance between false positives and false negatives by adjusting the thresholds.

VIII. Discussion

Comparison with type-based indirect-call analysis. Although type-based indirect call analysis can generate many false positives in large software, in theory, it can ensure soundness. This is essential for some special applications, such as control-flow integrity (CFI). However, many security applications do not require soundness but accuracy (balanced precision and recall). Most recently, Lu proposed the Range-Aware Multi-layer Type Analysis (MLTA) [27] in order to enhance the accuracy of indirect call target identification. Although this method holds the potential to improve the performance of security applications, our study (see §II) and evaluation results (see §VI) have shown that, due to the prevalence of cases where the scope of type-based approaches is exceeded, this method still falls short in providing precise assistance for program static analysis, such as inter-procedural taint analysis. More importantly, our system, which is designed to filter target functions based on abstract similarity, can provide complementary benefits to these type-based approaches, including the range-aware MLTA. By incorporating the strengths of both approaches, the performance of both our and their system can be further improved.

Comparison with CFI-based indirect-call analysis. In the realm of refining indirect call targets, several state-of-the-art approaches, including OS-CFI [22], τCFI [16], and PathArmor [42], utilize CFI techniques to refine indirect call targets during runtime. While these approaches benefit from dynamic analysis and can effectively analyze user space programs with high precision, they suffer from low coverage rates for large programs, particularly OS-level software, and cannot be easily applied to low-level programs, such as device drivers, without hardware support. For example, HFL [23] and DR.FUZZ [57], which are two recent kernel fuzzers, show that the code coverage for Linux kernel and its drivers is typically less than 10%, and sometimes even less than 1% for some drivers. Beyond this, as discussed in OS-CFI, such CFI-based techniques typically require hardware features, including Intel MPX, Intel TSX, and Intel PT. But, in contrast, our approach is based on static code analysis and does not require hardware support, making it suitable for analyzing large and low-level programs. Thus, while both our approach and CFI-based techniques aim to identify indirect call targets, they cater to different types of programs and have distinct security applications.

Path-based indirect-call analysis. Most of the existing indirect-call detection approaches aim to give all the possible target functions for a given indirect call. However, for some situations that require analyzing the specific execution paths, such as path-sensitive analysis, the target functions given by the current approaches are too general. For example, although, on average, we can reduce the number of cases for finding the correct call chain from 10^9 to 10^4, based on our results compared to type-based methods (see §VII-A), this is still not unique. The target function of the indirect call should be uniquely identifiable by giving enough preconditions and constraints. However, such path-sensitive indirect call analysis is out of this project's scope. We would like to regard it as interesting future work.

Limitation of current work and potential future works. First, finding the anchor functions is an unexplored problem, and none of the existing methods can handle this problem well. So in this project, we chose to combine the IR level information with the source code level information to find the anchor functions within the restricted scope. However, due to the lack of accurate matching methods from IR variables to variables on the source code, and the lack of accurate source code or pre-compiled code analysis tools, we cannot match the name of each function pointer with one hundred percent of accuracy. This part of the problem can be alleviated in the future by improving the precision of source-level value and name analysis. Such a tool is preferably compiler-based, which can make the analysis more accurate. In this way, it automatically extracts the names and initializers of all function pointers during the compilation process, thereby assisting GNNIC in associating function pointers with target functions.

IX. Related Work

Refining indirect-call targets. Recently a number of works on refining indirect-call targets emerge, including value-set analysis, point-to analysis, type analysis, and Neural Networks. Some previous works are working on refining the indirect-call targets on the binary level. For example, BPA [24] is a binary-level points-to analysis framework based on a block memory model, which can improve precision by 34.5% compared with other binary-level techniques. Icallee [59] is a Siamese Neural Network approach that uses NLP to embed the context of call instructions, reducing indirect call targets compared to other binary-level approaches like BPA and τCFI [16] but has better performance. However, such approaches are only tested on small programs, such as libraries, and are not scalable on large programs, such as OS kernels. Also, currently, the source-level solutions can still generate much better performance than the binary-level approaches, such as Icallee. Source-code level and IR-level pointer-based approaches, such as K-miner [14] and SVF [41], are also used to analyze function pointers. However, as we discussed, such approaches are typically not scalable for large programs or have a high false-negative or false-positive rate. Therefore, as we discussed, type-based approaches are typically the best choice for source-code level indirect-call targets identification. And moreover, multi-layer type-based solutions [28, 30] have been proposed in recent years, which improve the type-based approaches and have much better results. However, still, many function pointers do not qualify multi-layer type analysis, and our evaluation (see §VI) shows that for kernel-level large programs, GNNIC can still improve such state-of-the-art approaches a lot.

Measuring the similarity of code. Machine learning techniques are widely used in program and code similarity analysis. Code2Vec [2] can embed code pieces into fixed-length code vectors by aggregating the information of collected code paths in the abstract syntax tree (AST). Func2Vec [10] is based on a neural network, which can generate function vectors by random walking on the interprocedural control-flow graph of programs. FuncGNN [34] is a graph neural network trained on a labeled control-flow graph, which can be used to estimate the graph edit distance of the program. FA-AST [45] trains the GNN on a flow-augmented AST, which can catch the data and control-flow information of the program. Gemini [52] is a neural network-based approach to compute the embedding of functions based on their control flow graph at the binary level. All the above techniques are based on the program's control and data flow, which can be used in clone detection or predict semantic properties of the code snippet. Unlike these works, some machine learning models are also trained on program call graphs. Xu et al. [51] can detect Android malware, which leverages the natural language processing techniques to compute the embedding of the program by analyzing the whole call graph. DeepCatra [50] is designed to detect the malware behaviors of Android APPs. DeepCatra consists of a bidirectional LSTM (BiLSTM) and a GNN, which is trained on the features from the call trace of the program. Different from all these works, GNNIC focuses on the function call and types related information, representing the features of indirect-call target functions better.

X. Conclusion

In this paper, we introduce GNNIC, an innovative call graph construction tool that seamlessly combines program analysis and GNN to accurately identify indirect-call targets. This enables the creation of highly precise call graphs for large programs. GNNIC uses abstract-similarity analysis to match target functions based on the abstract behaviors they share. To realize this approach, we propose techniques, such as scoped unique-name matching, to identify anchor functions, and the use of a *representative abstraction graph* to incorporate diverse information about a function. The graph can be fed to GNN for training embedding models. Our evaluation results show that GNNIC significantly improves precision compared to existing state-of-the-art techniques. Additionally, we demonstrate that using function abstraction-based similarity analysis alongside precise call graphs can be effective in a wide range of security applications.

XI. Acknowledgment

We would like to thank the anonymous reviewers for their feedback and suggestions. This work was initiated during Qiushi Wu's internship at IBM Research and further finished upon his return to the University of Minnesota. Kangjie Lu and Qiushi Wu were supported in part by NSF awards CNS-1815621, CNS-1931208, CNS2045478, CNS-2106771, and CNS-2154989. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In *International Conference on Computer Aided Verification*, pages 158–163. Springer, 2005.

[4] R. Bavishi, H. Yoshida, and M. R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 613–624, 2019.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[6] Bootlin. The function `e1000e_check_for_fiber_link` and its description, 2022. URL https://elixir.bootlin.com/linux/v5.7/source/drivers/net/ethernet/intel/e1000e/mac.c#L459.

[7] R. Conroy. Sample size: A rough guide. *Retreived from http://www.beaumontethics. ie/docs/application/samplesizecalculation. pdf*, 2015.

[8] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[9] C. Data61. Stellargraph machine learning library. https://github.com/stellargraph/stellargraph, 2018.

[10] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*, 2018.

[11] P. S. Foundation. random — generate pseudo-random numbers, 2023. URL https://docs.python.org/3/library/random.html.

[12] L. Fu, S. Ji, K. Lu, P. Liu, X. Zhang, Y. Duan, Z. Zhang, W. Chen, and Y. Wu. Cpscan: Detecting bugs caused by code pruning in iot kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 794–810, 2021.

[13] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[14] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[15] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020.

[16] J. Grossklags and C. Eckert. τcfi: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050, page 423. Springer, 2018.

[17] S. group. Syzbot fuzzing logs, 2022. URL https://syzkaller.appspot.com/upstream/.

[18] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017. URL https://snap.stanford.edu/graphsage/.

[19] H. Han and S. K. Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[20] C. T. Inc. Dynamic call tracking method based on cpu interrupt instructions to improve disassembly quality of indirect calls, 2020.

[21] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[22] M. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang. Origin-sensitive control flow integrity. In *USENIX Security Symposium*, pages 195–211, 2019.

[23] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[24] S. H. Kim, C. Sun, D. Zeng, and G. Tan. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium, NDSS*, 2021.

[25] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. Cab-fuzz: Practical concolic testing techniques for cots operating systems. In *USENIX Annual Technical Conference*, pages 689–701, 2017.

[26] C. Liu, Y. Chen, and L. Lu. Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[27] K. Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1610–1624. IEEE Computer Society, 2023.

[28] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[29] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932, 2016.

[30] K. Lu, A. Pakki, and Q. Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019. URL https://github.com/umnsec/crix.

[31] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, Vancouver, BC, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry.

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

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[34] A. Nair, A. Roy, and K. Meinke. Funcgnn: a graph neural network approach to program similarity. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[35] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[36] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *USENIX Security Symposium*, pages 729–743, 2018.

[37] J. Pan, G. Yan, and X. Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *USENIX Security Symposium*, pages 149–165, 2017.

[38] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, pages 167–182, 2017.

[39] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[40] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[41] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[42] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940, 2015.

[43] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[44] S. Vijayarani, M. J. Ilamathi, M. Nithya, et al. Preprocessing techniques for text mining-an overview. *International Journal of Computer Science & Communication Networks*, 5(1):7–16, 2015.

[45] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020. doi: 10.1109/SANER48275.2020.9054857.

[46] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with kint. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, 2012.

[47] V. M. Weaver and D. Jones. perf fuzzer: Targeted fuzzing of the perf event open () system call. *UMaine VMW Group, Tech. Rep*, 2015.

[48] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[49] Q. Wu, Y. Xiao, X. Liao, and K. Lu. Os-aware vulnerability prioritization via differential severity analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 395–412, 2022.

[50] Y. Wu, J. Shi, P. Wang, D. Zeng, and C. Sun. Deepcatra: Learning flow-and graph-based behaviors for android malware detection. *arXiv preprint arXiv:2201.12876*, 2022.

[51] P. Xu, C. Eckert, and A. Zarras. Detecting and categorizing android malware with graph neural networks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 409–412, 2021.

[52] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[53] A. A. Younis, Y. K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8. IEEE, 2014.

[54] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *Usenix Security Symposium*, pages 363–378, 2016.

[55] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger, et al. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *2022 Network and Distributed System Security Symposium*, 2022.

[56] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, 2019.

[57] W. Zhao, K. Lu, Q. Wu, and Y. Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *NDSS*, 2022.

[58] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[59] W. Zhu, Z. Feng, Z. Zhang, C. Zhang, Z. Ou, and M. Yang. icallee: Recovering call graphs for binaries. *arXiv preprint arXiv:2111.01415*, 2021.