applied sciences

Article

RepFTI: Representation-Fused Function-Type Inference for Vehicular Secure Software Systems

Xiaoyu Yi , Gaolei Li *, Jianhua Li and Ao Ding

School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China; yxy1234@sjtu.edu.cn (X.Y.); lijh888@sjtu.edu.cn (J.L.); ao_ding@sjtu.edu.cn (A.D.)
* Correspondence: gaolei_li@sjtu.edu.cn

Abstract: To enhance the security of vehicular software systems, inversely identifying the underlying function types of binary files plays a key role in threat discovery. However, existing function-type inference (FTI) methods can only provide a suboptimal performance because of splitting binary files into multiple sub-blocks as inputs, which results in breaking the program context logic and complete data dependency. To solve this problem, we propose a novel representation-fused function-type inference (RepFTI) framework for secure vehicular software systems. First, the RepFTI learns semantic representations of assembly codes and then extracts node representations in the function call graph by the multi-head attention mechanism of Graph-Attention Transformer (GAT) models. Second, the RepFTI fuses these representations to accurately infer the function type. With RepFTI, the specific limits of in-vehicle software will be bypassed, which proposes a promising direction for other work that relies on reverse engineering to improve software security.

Keywords: function-type inference; mutli-representation; code semantic learning; deep learning

check for updates

Citation: Yi, X.; Li, G.; Li, J.; Ding, A. RepFTI: Representation-Fused Function-Type Inference for Vehicular Secure Software Systems. *Appl. Sci.* **2024**, *14*, 4502. https://doi.org/10.3390/app14114502

Academic Editor: Luis Javier García Villalba

Received: 17 April 2024
Revised: 17 May 2024
Accepted: 17 May 2024
Published: 24 May 2024

1. Introduction

Vehicular software systems have been undergoing major disruptive structural transformations. The major transformation is the vehicle–road–cloud collaboration, which causes many applications with many new features, and many sensitive data are introduced [1]. Furthermore, compared with the previous vehicular software system, these new features make the vehicular software design have many service interfaces, which greatly expand the attack surfaces of the vehicular software [2–6]. And, the threat of risks to in-vehicle software systems is gradually increasing [7–9]. The available format of in-vehicle software often appears in binary code to meet the security required by in-vehicle software manufacturers.

Binary code is often available for the software in vehicles. At the same time, the compiler does not have enough language-level semantic information, such as the function type, the original data structure, and so on, in the process of compilation. More specifically, recovering the semantics of the function is important for applications, such as bug searching [10–12], code clone detecting [13,14], patching and repairing [15–17], and patch analysis [18–20]. And, the function-type inference task is a classic research direction in recovering the semantics of the function. The function-type inference methods often use the reliable disassembly of instructions to recover the control flow, the data structure, or the function semantics to recognize the function type. Additionally, this recovered information with higher-level semantic descriptions can provide more specialized analysis tasks.

To ensure the availability and security of vehicular software systems, software security engineers need to make different decisions during the software life cycle. The two classic phases need the function-type inference to improve the security of the software in vehicles. In the test phase of software in vehicles, due to the rapid iteration of the software, the software engineers need a more accurate tool or strategy to find the target function that matches the semantics of vulnerability or the key function. Similarly, in the software operating, the vulnerability or the code defects of the software make the vehicles lose connection or control, so a function-type inference tool is needed to detect suspicious functions in the execution process early. In the field of in-vehicle software, since the new modules of software and the new protocols of the platform level are proposed, the service-oriented architecture (SOA) that has abstractions and standardized service interfaces decouples the application development and underlying platform support [1]. However, the different vehicle manufacturers have a heterogeneous abstraction level and standardized interfaces, which cause a new SOA to map a new function-type inference strategy. More specifically, the service functions in the underlying platform are packaged, which causes the detailed semantics of the function of software in vehicles to be lost. With the function-type inference (FTI), software engineers can not only implement reverse engineering but can also promote the development of other related fields (etc., function boundary detection, code-similarity detection, and vulnerability detection). An accurate FTI model can improve the efficiency of bug searching, the values set inferring, and fuzzing.

Deep learning has been perceived as a promising technique to implement intelligent FTI tasks. It usually needs to complete two following steps. The first step is to determine the analysis granularity (e.g., the dataset organization mode). For example, Shin et al. [21] proposed to use natural language processing methods to learn the semantics of assembly codes under static analysis scenarios. The second step aims to determine representation methods according to the application requirements. Existing representation methods can be mainly divided into two categories: (1) text-based methods [22–25], which mainly extract features from assembly codes and control flow information that are reserved from binary files, and (2) graph-based methods [26–29], whereby software engineers can build graph-representation models at the function, basic block, instruction, and other levels. For example, a control flow graph-based embedding method is proposed in [28] for efficient binary code-similarity detection. And also, the authors in [30] model the call relationship between basic blocks as a graph network to detect software vulnerabilities. But, they do not fuse text semantic features and graph features, which leads to the absence of global data dependencies [31].

In the software of vehicles, the function-type inference has two sub-problems: recovering the semantics within the function and the control flow between functions. At first, the abstract-level functions are often met when we try to trace the execution process of in-vehicle software, but the general dynamic binary analysis tools are hardly useful to identify the abstract-level interface. And then the semantic feature can only find the similarity execution process within functions, but how can we filter some similarity local execution processes with different global dependency relationships? These are mainly problems in existing works. In this work, we extract the features of functions and recognize the function type by the semantics and the control flow of functions. Our starting point is a list of assembly instructions at the function level, and the goal is to produce the observable function type without any compilation information to obtain early and clear findings of the vulnerabilities and code defects.

Our Approach. In this paper, we propose a novel representation-fused function-type inference (RepFTI) framework, which focuses on using multi-representation fusion to recover global data dependencies that were destroyed during the model-training process. The RepFTI mainly consists of the following two parts: (1) generating semantic representations and graph representations, and (2) fusing multiple representations to reconstruct the whole data dependency. For semantic representations, we generate word embeddings by refining the token classification and modifying the embedding model. For node representation, we traverse each node in the function call graph and extract the call relationship feature between the target node and its neighbor nodes as the node representation. Corresponding to the reconstruction process of complete data dependency, short-span data dependencies are recovered using a Bidirectional Long Short-Term Memory (Bi-LSTM) network, and long-span data dependencies are recovered with a GAT model.

2. Related Work

In this section, we introduce the roles of FTI in improving the security of decision making during the vehicular software system lifecycle, as shown in Figure 1. By comparing representation strategies, we also clarify the current challenges and unsolved problems of existing FTI approaches.

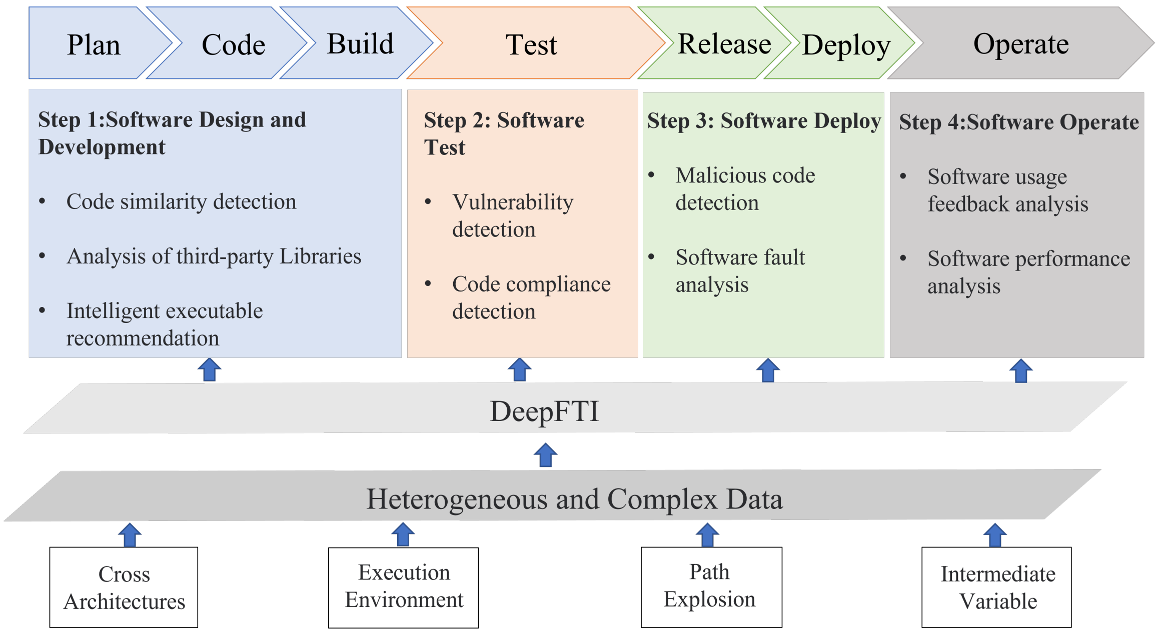


Figure 1. Due to the heterogeneous execution environment, various compilers, and frequent code reusing, a large-scale software system becomes more and more unobservable. With RepFTI, the whole lifetime of a software will be more transparent, controllable, and trustworthy.

2.1. FTI for Software Design Security

Due to the birth of new technologies of vehicles (e.g., service-oriented architecture) and software modularity, the design and implementation of vehicular software looks more like a process of compositing various types of functions, which causes the loading of reusable binary files to be a popular method during software implementation. Automated FTI strategies can help improve the efficiency of the software design and implementation from three main directions, including code-similarity detection, a reliability analysis of third-party libraries, and executable program recommendations. The main principle of automated FTI is choosing one type of representation for the target binary file and designing neural-network-based high-dimensional feature analysis strategies. Therefore, it can utilize the high-dimensional space to significantly optimize the inference efficiency and precision of complex software. For example, InnerEye [31] takes the assembly code as the representation and the basic block as the analysis granularity. The pre-trained model transforms key semantic features of the assembly code as feature vectors, which helps the neural network understand the logic between instructions. But, a significant amount of code comparison operations and logical analysis tasks will take too many resources.

2.2. FTI for Software Security Testing

Prior to software deployment, many testing samples are required to detect vulnerabilities in mitigating the threat behaviors of software. Existing testing methods in the industry majorly relying on string matching require manually preparing specific test samples, which may overlook some errors or be bypassed by malicious codes. Different from these methods, deep-learning-based FTI methods mainly focus on extracting implicit features to identify specific functions in target vehicular software systems [32]. For instance, Asm2Vec [22] made progress in analyzing the relationships among semantic features of assembly code and assigning different weights to these features, taking the instruction as a minimum analysis unit.

2.3. FTI for Secure Software Deployment

Both users and software developers focus on the security and stability of software during the deployment phase. Service providers hope to have a deep understanding of the specific operating procedures within the software to further improve the software-execution efficiency. However, vehicular software developers do not want to open the source code because they cannot guarantee that the users who obtain the open source code are trustworthy to ensure the security of the software. Therefore, software developers mostly release software systems in the form of packed executable programs. But, the packed executable program needs a lot of manual analysis to recognize the threat behaviors and understand the program code logic. To address this issue, many methods have introduced multi-representation neural network models into FTI. For instance, Z. Yu et al. [33] not only capture assembly code features but also adopt a convolutional neural network on adjacency matrices to extract sequential information that improves the model's power for graph-similarity detection, which can provide more insights for understanding the function logic of binary files. Similarly, PalmTree [34] mainly utilizes control flow and data flow to train an accurate instruction-embedding model for downstream tasks.

2.4. FTI for Software Maintenance Security

In the real service environment, it is difficult for executing vehicular software to interrupt the execution in order to detect threatening behavior, and existing software maintenance methods mainly depend on performance testing records and user feedback. However, large-scale software has various internal packaging components and numerous execution paths. Only individual vulnerabilities can be repaired based on testing records and user feedback, and it is impossible to comprehensively and continuously observe the software status. A dynamic analysis refers to monitoring all operations of the target binary code, observing the code-execution process, and collecting various data during the code-execution process when running a software program in a controlled environment. Training the neural-network-based FTI model based on a dynamic analysis has great potential to provide more real-time and accurate software maintenance services. For example, the BINGO-E [35] uses a dynamic analysis-based strategy to capture complete semantic representations that include the target function's semantics, code semantics of the called libraries, and user-defined functions' semantics. But, due to the unsettled challenges of dynamic analysis technologies, like incomplete code coverage and an unreliable execution path track, this kind of method is not yet taken seriously.

2.5. Motivation and Challenges

Although the application of FTI has significantly improved the security of software design and implementation, software testing, software deployment, and software maintenance, it also faces many challenges and unsolved problems. Specifically, in the environment of in-vehicle software, these problems and challenges can be amplified by some new features of in-vehicle software. Table 1 summarizes and compares existing approaches based on the representation types, the splitting granularity of target binary files, the encoding schemes, and the model architecture.

Table 1. Comparisons between existing FTI strategies.

Name	Raw Byte	Information Graph	Assembly Code	Number of Representations	Granularity
ClearView [36]	✗	✗	✓	Single representation	Basic block
Tough call [37]	✗	✗	✓	Single representation	Function
TIE [38]	✗	✗	✓	Single representation	Function
EKLAVYA [39]	✗	✗	✓	Single representation	Function
Order matters [33]	✗	✓	✓	Double representations	CFG, basic block

2.5.1. Incomplete Data-Dependency Reconstruction

Assembly code is the closest high-level language to machine code and displays various low-level information about a software program in a fine-grained way. However, when the analysis target is a complex software system, the corresponding assembly language sequences are very long and chaotic so that its data dependencies may be broken up and hard to completely reconstruct using FTI models. To address these issues, existing FTI strategies tend to use neural networks to extract high-dimensional features from the chaotic assembly code and recognize function types. But, to fit the fixed input size of neural network models, a long assembly code has to split into multiple code fragments, leading to losing data-dependency issues and unreliable FTI. To the best of our knowledge, there are no studies that focus on quantifying the impact of data dependency on the security of software systems. To bring out the problem of data-dependency loss, we provide a real case as follows. The simple $printf$ function with the C-language style involves executing the $printf$ function as well as the $write$ function and the system call of $write$. This case relates to many kinds of data dependencies, including a storing process of system call numbers, an executing process of an interrupt instruction (int 0X80) for transiting the user state to the kernel state, a finding process of the system call entry address sys_printf, and an interrupt return process. When we train a neural network to generate an FTI model, all data samples are loaded in a random way rather than in a sequential way, which means that the native order of the assembly code will be disorganized. Therefore, the data dependency during the execution process of $printf$ can not be learned completely by the FTI model.

2.5.2. High Dynamic Tracking Loss Rate

Dynamically tracking the binary file's execution path often leads to encountering some hidden function calls (such as system calls, function pointers, and inline functions), which have not been linked to the instruction being executed using the pointers. Meanwhile, to improve the software-execution speed, some specific jump pointers that can alter the CPU control permission may have been redirected to obscure the entry points and even the overall structure of the function. These factors increase the difficulty of accurately calculating target function boundaries. But, to construct function-type labels, it is necessary to design a strategy to discover the boundaries of each function as clearly as possible when tracking the target software-execution process. The hidden function calls and the jump pointers may bring a high dynamic tracking loss rate.

3. Proposed RepFTI Framework

In this section, we will introduce the proposed RepFTI framework in detail, as shown in Figure 2, which mainly includes a novel dataset-generation method, a semantic learning model, and a graph-learning model. With RepFTI, more function types can be accurately identified because the data dependencies of target vehicular software systems can be reconstructed more completely.

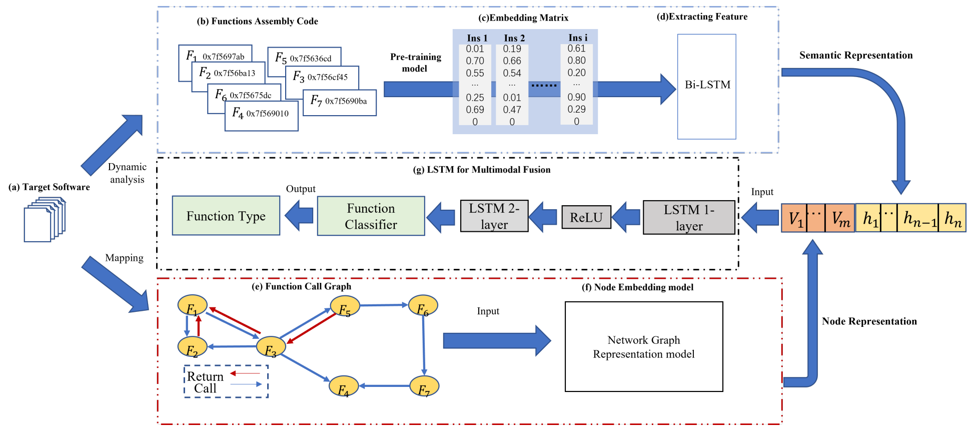


Figure 2. The RepFTI is divided into two parts: (1) extracting the multi-representation features, and (2) inferring the function types. In the first part, a novel dynamic tracking strategy is proposed to obtain the semantic representation within functions and the node representation in function call graphs at one-time execution, respectively. And then the semantic learning model extracts short-span data-dependency features within a function, while the graph-embedding model extracts long-span data-dependency features among functions. In the second part, the multi-representation feature fusion model fuses the above two kinds of data dependencies.

3.1. Dataset Generation

In this section, we begin to solve the challenges of the high dynamic tracking loss rate from Section 2.5.2. To reduce the risk of losing the pointer that is linked to the currently executed instruction, we propose an innovative approach that uses a stack and queue data structure to record the program stack. The queue stores function attributes with the address as the key and the full-function call order. The stack simulates the program stack-execution process, which records the *_ebp* and *_esp* register values, and target addresses of a call instruction to closely track the target binary execution process. When calling the *%ret* instruction, the tracker compares the most recently recorded *_ebp*, *_esp* register value in the stack with the address to which the return is jumping. If the two values are equal or the difference between the two values falls within 15 bytes (the maximum instruction length), that value is recorded. Otherwise, the jump target address is recorded. Similarly, when calling the *%call* instruction, the tracker compares the most recent *_ebp* and *_esp* register value with the address to which the *%call* instruction jumps and follows the same rules as above. Through the above rule, all adopted values are recorded in the dictionary. This rule ensures that the tracked call and return addresses are accurate and removes some factors that affect the construction of the function call graph.

To meet the requirements of RepFIT for constructing function labels, the mapping of the function name and assembly is the next problem that must be solved. Typically, the initial preference is to dynamically obtain the function name. However, the process of querying the *plt* and *got* table makes dynamic analysis obtain the function names complexly. Subsequently, we found that Pintools [40] refers to *Routine Object* : *RTN* [41] (a kind of granularity defined by Pin) during both static analysis and dynamic analysis, and it assigns a unique symbol to *RTN*. By our verification, *RTN* is an equivalent function (we compare *RTN* assembly codes with assembly codes of functions of the same symbolic name at static analysis). The unique symbol of the *RTN* can be used as an index for function names and function assembly codes. After finding a unique symbol of the function, the function call graph can integrally be constructed with steps (b) and (e), as shown in Figure 2. When calling a *call* instruction, a directed edge is generated from the node of the calling function to the called function. When calling a *ret* instruction, in order to find the call instruction corresponding to the return instruction, the adopted address is traversed through the dictionary. If there is no corresponding value, the return instruction is discarded. The above is the complete process of mapping function names and function assembly codes.

3.2. Semantic Learning Model

To solve the challenge that was introduced in the incomplete data-dependency reconstruction sub-section, we propose a multi-representation fusion strategy that combines semantic representation within functions with function call graph representation. Steps (c), (d), (f), and (g) in Figure 2 illustrate the design of the fusing multi-representation process, which consists of four main components: extracting semantic features from the assembly code semantic representation divided into extracting semantic features within instructions and extracting semantic features between instructions (c) (d), extracting function call graph features from function call graph representation (f), fusing semantic representation and network graph representation (g), and inferring the target function type. The embedding model of RepFTI is based on the PalmTree [34], and it adds the following important design considerations. We first introduce the extract assembly code semantic feature vector from semantic representation.

The process of extracting semantic features from assembly code semantic representations can be divided into two parts: (1) extracting semantic features within instructions, and (2) extracting semantic features between instructions. In the first part, it is necessary to design an accurate instruction-embedding model, where the instruction should be treated as a sentence and the operands and operators are treated as words. Step (c) describes the instruction-embedding process that transforms each assembly instruction into a feature vector. On the basis of PalmTree, our embedding model can efficiently learn the semantic feature within instructions for more operators and operand tokens to be added. New tokens are refined as the operand part and the operator part. Given an operand, it will be considered as an address with a special token $[constant]$ if its length is longer than three digits; otherwise, it will be considered an immediate number. Meanwhile, the mentioned address can be further divided into the absolute address and relative address, with $[ab\_addr]$ and $[re\_addr]$, based on the approximate address segment in the memory layout where the data are stored. For the operator part, we focus on jump instructions that can change the control flow and have a great impact on the data dependencies of the target software. In this case, the current address, target address, and function name of these instructions are recorded to extract semantic features. This enhancement contributes to the refinement of learning data dependencies within functions.

To achieve an effective embedding model, RepFTI randomly masks the tokens and uses the improved PalmTree model to predict the real token, in which the context relationship in the assembly code is learned. The improved PalmTree model, built upon BERT, is characterized by bidirectional connectivity and is structured into multiple layers (e.g., encoder and decoder) within transformer units. Therein, the encoder is trained to predict the masked token. The whole training task consists of two sub-tasks. The first sub-task is to obtain a Masked Language Model (MLM), which understands the internal structure and data flow of assembly instruction [42]. The second sub-task aims to jointly adjust the embedding model and the instruction order prediction model to learn the data dependencies between instructions.

In the second part, we use an advanced Bi-LSTM model based on [43] to extract the semantic feature between instructions. To prepare standard inputs for the advanced Bi-LSTM model, the assembly code is transformed into a feature matrix using pre-trained word embeddings. Moreover, step (d) compresses the feature matrix into a unified vector to infer function types.

3.3. Graph-Learning Model

The short-span data dependency is reconstructed by assembly code semantic learning, but the complete data dependency of target vehicular software systems needs long-span data dependency between functions. However, the long-span data dependency has a significant difference from the short-span data dependency. Subsequently, we introduce the strategy of leveraging the function call graph features to reconstruct long-span data dependencies.

The function call graph of an execution path views the data-processing logic that displays a group of states around the target function. In other words, a function is denoted as the node, and the call relationship between different functions is denoted as an edge. The depth-first searching algorithm for a basic graph analysis focuses on retrieving the upstream and downstream information of the target function, but it can only find the source of the data related to the target function for higher-dimension features of function call graphs that can not be extracted. Therefore, we create a special GAT model based on [44] to extract the high-dimensional features of these function call graphs. For the specially designed model, the GAT layer generalizes the attention model on the function call graph and other attributes of the function node. By stacking layers, nodes gain the ability to attend to the features of their neighboring nodes. When we analyze a function, the relationship between the target node and neighbor nodes is the key feature. Then, each node of the function call graph represents functions, and the edge stands for the directed call relationship.

Thus, the graph-learning process combines the call relationship with the execution states (e.g., the call address, call depth, and so on) of target binary files, which improves the long-span data-dependency feature between functions. A real example is shown in Figure 3, whereby we consider a single-node training process that extracts the long data-dependency feature of the target function node. Each function node stores five attributes in the nodes, including $node\_id$, $node\_feature$, $head\_node$, $tail\_node$, and $edge\_feature$, in a matrix way. Specifically, $node\_feature$ is the main attribute of the node, and others are assistant attributes. These matrices are GAT stand input. The learning process of GAT through build-the-block layers and stack-block layers constructs a graph-attention layer that produces each node, like n_3, which are feature vectors according to graph structure matrices and neighbor node attributes. Then, to improve the generalization ability of the learning graph model, we use the multi-head attention mechanism of GAT to join the independent nodes featured above, in which this mechanism uses random sampling node features from the graph-attention layer and joins them to obtain target node features as the long-span data dependency of target functions.

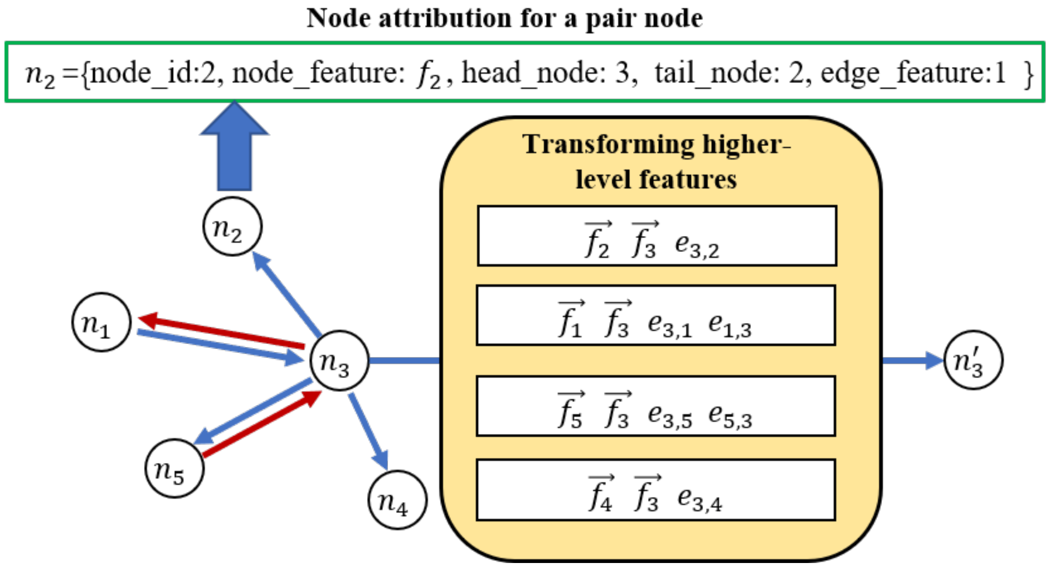

Node attribution for a pair node
n_2 ={node_id:2, node_feature: f_2, head_node: 3, tail_node: 2, edge_feature:1 }
Transforming higher-level features
$\vec{f_2}$ $\vec{f_3}$ $e_{3,2}$
$\vec{f_1}$ $\vec{f_3}$ $e_{3,1}$ $e_{1,3}$
$\vec{f_5}$ $\vec{f_3}$ $e_{3,5}$ $e_{5,3}$
$\vec{f_4}$ $\vec{f_3}$ $e_{3,4}$
n_1
n_2
n_3
n_4
n_5
n_3'

Figure 3. Graph structure for the call relationship of a single target function. Each node includes five kinds of attributes, while each edge stands for the call relationships between different functions.

The semantic representation's feature vectors are produced by the semantic learning model, while the graph representation's feature vectors are generated by the graph-learning model. With these two kinds of representation features, both shot-span and long-span data dependencies are extracted, which is essential to improve the accuracy and robustness of FTI. The representation fusion is shown as step (g) in Figure 2.

To fuse the above two kinds of representation features, we concatenate them into the same vector instead of mathematical operations to avoid the feature loss problem. Then, to obtain the complete data dependency of each function and reliable prediction results, we use a cascaded Long Short-Term Memory (LSTM) model to cyclically learn the concatenated feature vector. In this model, the first LSTM network reconstructs the concatenated feature vector as an implicit variable that contains complete data dependency. The second LSTM network cyclically updates the implicit variable according to a well-labeled function call order. Through the above design, function-type inference results with high confidence can be achieved.

4. Evaluation

To check that the RepFTI can improve the observability of the target software, we develop and implement a comprehensive evaluation framework, encompassing both intrinsic and extrinsic assessments. This section introduces our evaluation framework and outlines the experimental configurations, followed by detailed discussions about experimental results.

4.1. Experimental Setup

Datasets. We select four mainstream tool libraries (these libraries are common on vehicular computers) of vehicular software systems to make a benchmark dataset for our model and promote the evaluation results' generalizability. This dataset includes different versions of Openssl, Curl, and Busybox on the x86-64 platform, as well as binary files compiled with different optimization levels (O0, O2, and O3). The whole dataset consists of 3100 executable paths, 40 million functions, and 2.1 billion instructions in total. This dataset has code coverage above 80%.

Hardware Configuration. Due to a lack of various vehicles, to test as many vehicle environments as possible, we build a virtualized host-based android for the CAN communication. All the experiments of the train phase and inference phase are conducted on the virtualized E/E architecture of vehicles based on android cars (X86 architecture) that are run on a Xeon(R) 8259CL CPU 2.50 GHz $\times$ 16, one GTX 1060 GPU, 6 GB memory.

4.2. Evaluation Methodology

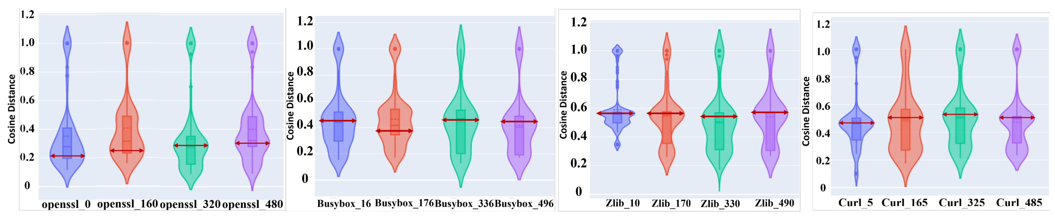
We utilize the intrinsic evaluation to prove the observability of each representation feature and the extrinsic evaluation to validate the effectiveness of the RepFTI. In the evaluation of word-embedding methods, the intrinsic evaluation involves comparing the generated embeddings with human assessments [45]. Similarly, for assembly code semantics and function call graphs, the intrinsic evaluation depends on the observation dimension of the analysis objectives.

4.2.1. Intrinsic Evaluation

Since each function possesses an immutable data-processing logic, the given function has great immutability. This provides the possibility of observing the accuracy of the semantic representation. By calculating the similarity between the target function's semantic features and the given baseline, we can evaluate the observability of RepFTI in accurately extracting semantic features. For the key step of selecting the evaluation baseline, the testing set of ReliFTSI is to divide a function into 10 instruction fragments, and we calculate the mean of 10 instruction fragment feature vectors. Note that the split functions belong to the same function identification and come from different execution paths. Then, we select several feature vectors with a better clustering effect in these mean values as evaluation baselines in a target function.

The cosine distance measures the difference between two vectors in direction, not in real distance. When a pair of texts have a large length difference in similarity but similar content, if the word frequency or word vectors are used as features, their Euclidean distance in feature space is usually large. If cosine similarity is used, the angle between them may be small, so the similarity is identified as high. So, the similarity of text tasks is often determined by cosine distance. In Figure 4, we calculate the cosine distance between these functions and these baselines separately. In accurate detection, we design 200 different execution paths for each tool library to be tested and display accurate detection results to build box plots. Figure 4 shows the accurate detection result of some main functions. The X-axis represents the index of batches in this test (each batch contains 64 functions) and the Y-axis represents the cosine distance values. More specifically, each cosine distance signifies the proximity between the unknown function and the baseline eigenvector. The widths indicate the number of functions meeting this criterion, with red arrows highlighting the broadest location within the box. The results of the sample show that the cosine distance of more than 83.7% of the instruction sequence pairs in the test set can be controlled below 0.6. This can prove that there is a strong accuracy when extracting the semantic representation within functions.

Based on the immutableness of functions, they can also fit the function call relationship. To observe the reliability of the function call graph representations, this primary detection strategy is observing the clustering performance of representation that comes from the same executable file and different execution paths, as shown in Figure 5. This figure is a TSNE graph that displays the clustering result between function call graph-representation features. The different color dots in Figure 5 are two-dimensional representations of high-dimensional vectors that are function node features in function call graphs. And, the different colors of the dots map to a kind of function-calling type. The dots with the same color are clustered together, and the dots with different colors have distinct boundaries, which proves that the function-calling type can be recognized according to the function node feature clearly. The bivectors in Figure 5 come from the transformation of the feature vectors of a function call graph representation. As shown in Figure 5, nodes of different colors represent distinct representation features, and they show a great clustering performance among nodes of the same representation feature. And then we show the calling addresses and the called addresses of part nodes, which prove that there are call relationships between these nodes.



(a) Cosine Distance of Most Samples are 0.15--0.30 under Different Execution Paths

(b) Cosine Distance of Most Samples are 0.30--0.50 under Different Execution Paths

(c) Cosine Distance of Most Samples are 0.50--0.60 under Different Execution Paths

(d) Cosine Distance of Most Samples are 0.47--0.55 under Different Execution Paths

Figure 4. Immutability of data-processing logic under semantic presentation scenarios. Four kinds of tool libraries and random execution paths are used in our experiments. Each sub-figure corresponds to one tool library, which displays the cosine distance between the semantic feature vector of the same function in different execution paths and baseline feature vector. The sub-figure contains 640 functions per batch. The x coordinate indicates the index of batches (each batch contains 64 functions). The y coordinate represents the cosine distance values. The width of a y coordinate across the box plot indicates the amount of instruction segment pairs that have this cosine distance. The wider the distance, the more functions there are.

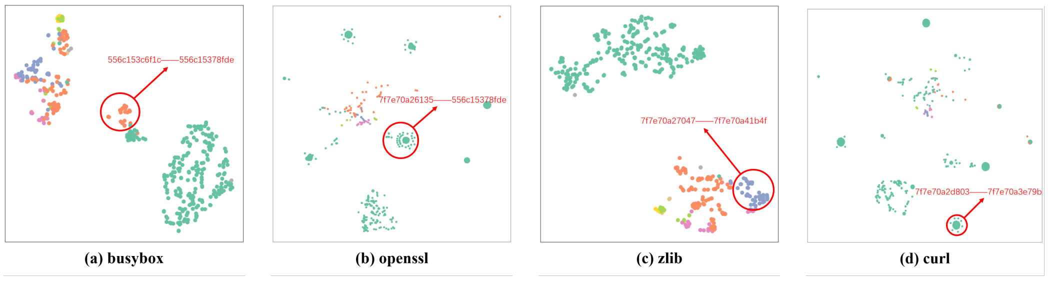


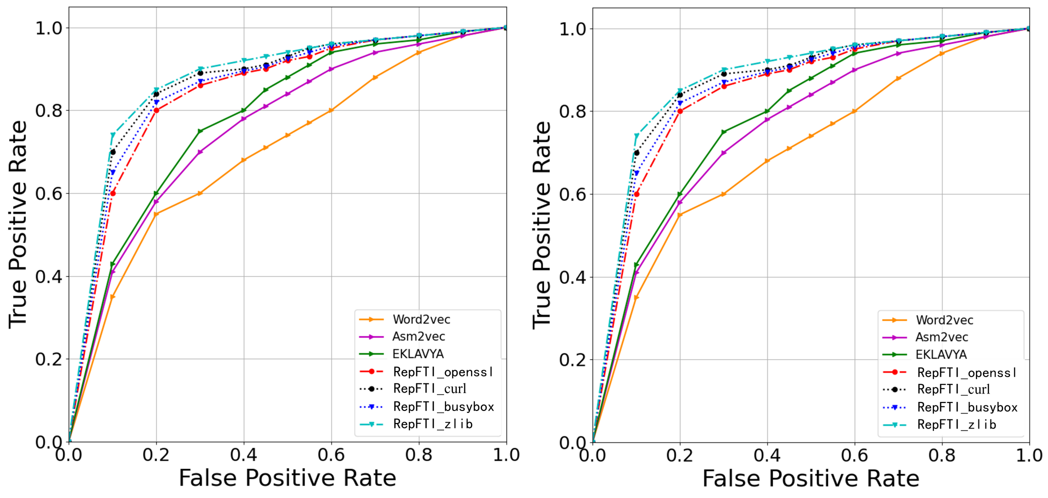
Figure 5. The observability of function call relationship. We construct a TSNE figure for each tool library according to the feature vector of function node representation. The framed parts are explicitly labeled with the range of source addresses for there is a direct or indirect call relationship between their corresponding functions. Dots with different colors represent different function call relationships, and it can be found that learning the difference between these function call relationships is easy.

4.2.2. Extrinsic Evaluation

To show the effectiveness of RepFTI, we assess the performance of RepFTI alongside other similar methods using a dataset generated from dynamic analyses of four tool libraries under compilation optimization level O2. At the same time, we also use completely different execution paths from the training set as the test set. Other execution parameters are consistent with common Settings. In Figures 6 and 7, we compare the impact of two execution paths, high compilation optimization and no compilation optimization, on RepFTI and can see that the execution path that retains more information has better recognition accuracy. At the same time, it shows that RepFTI has high accuracy and strong stability when detecting different software or tool libraries with the same test set as the input. Compared with the existing work, the recognition accuracy is significantly improved.

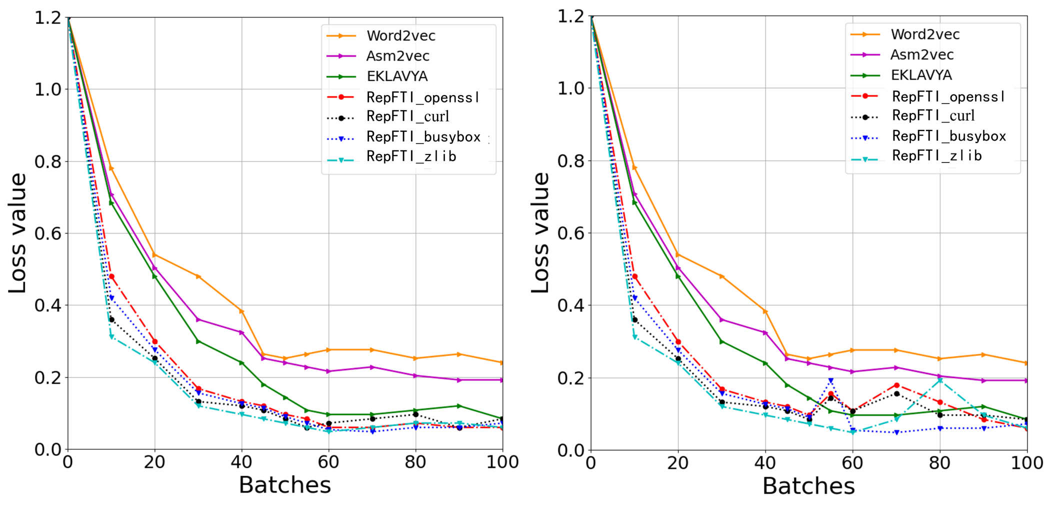
Due to the limited computing resources of vehicles, the in-vehicles software requires high transparency for deployed security modules. We check the time consumption of the two main parts of RepFTI, as shown Figure 8, including the semantic-representation-recognizing module and the graph-representation-recognizing module. We record a query of a single module of RepFTI 1000 times and calculate their average values. And then these time-consumption recordings of related current works are collected as baseline data to clearly compare the time efficiency, which includes GMNs [29], HGMN [46], and HGNN [47] as graph-representation-based function-type inference and InnerEye [31], Asm2Vec [22], and EKLAVYA [39] as semantic-representation-based function-type inference. Regarding the results of the evaluation, they basically met our expectations. Since the graph-representation recognizing takes on auxiliary work, the structure of the function-calling graph does not include more complex data. The graph-representation module of RepFTI does not consume much time, but the semantic-representation module needs to learn and recognize most assembly instructions, which makes the result of the semantic-representation module similar to other existing works.

To further test the effectiveness of the single representation of RepFTI, we close the semantic-representation model and the graph-representation model to record the corresponding accuracy of the function-type inference. To test the effectiveness of function-type inference based on the semantic-representation model, we select current works as a baseline to clearly show the performance of the semantic-representation model of RepFTI. Therein, these current works also use semantic representation to recognize the different parameters of the function, like the input type, the core logic, and so on. Therein, we select 1000 execution paths that can cover most codes in each library, which are used as input parameters for this evaluation experiment. Then, in the same way, we also find similar current works about using a graph-representation model and complete RepFTI (semantic + graph) to infer the function type, which helps us test the effectiveness based on graph representation and the effectiveness based on RepFTI. As shown Table 2, the experiment result is as expected. Using graph representation to recognize the function type does not have excellent results, because the functions with a similar call graph structure could be different functions with high probability. Although using semantic representation to recognize function type has promising accuracy, the standard deviation is higher than other current works. This status shows that the recognition of function types only by the semantic-representation model has high instability.



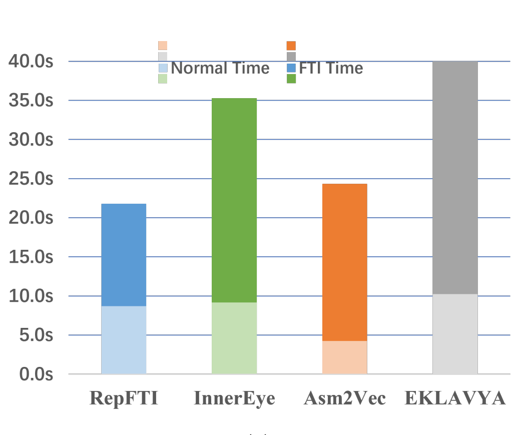
(**a**) (**b**)

Figure 6. Comparisons to existing methods on ROC curves. (**a**) The left shows that our proposed RepFTI outperforms existing methods in the case of low compilation optimization levels: O0. (**b**) When the compilation optimization level is raised up to O2, the RepFTI can still keep its advantages against existing methods.

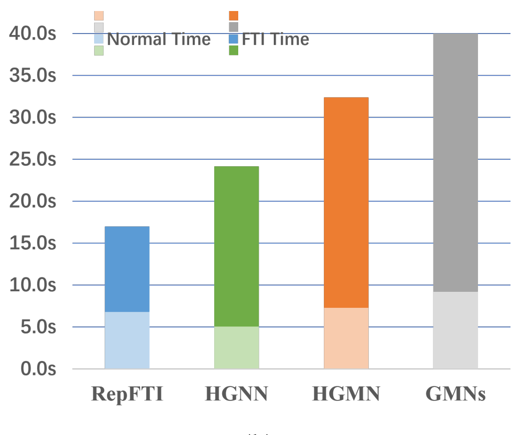


(**a**) (**b**)

Figure 7. Comparisons to existing methods on loss values. (**a**) The left shows that our proposed RepFTI outperforms existing methods in the case of low compilation optimization levels: O0. (**b**) When the compilation optimization level is raised up to O2, the RepFTI can still keep its advantages against existing methods.



(**a**)



(**b**)

Figure 8. Comparison of single module on the time efficiency. To show the transparency of RepFTI, in 1000 queries, the averaged time consumption of the semantic representation recognizing and the graph-representation recognizing are recorded. At the same time, these semantic-recognizing strategies (GMNs [29], HGMN [46], HGNN [47]) and graph-recognizing strategies of current works become baseline in this ablation study. (**a**) The module of recognizing semantic representation is shut down. (**b**) The module of recognizing graph representation is shut down.

Table 2. The effectiveness of function-type inference under single representation and multi-representation model. The results of the single semantic-representation model, the single graph-representation model, and both representation models are shown. Therein, the single representation model of RepFTI tests four libraries, and the *DDFSI_B*, *DDFSI_C*, *DDFSI_F*, and *DDFSI_Z* denote the *Bison*, *Curl*, *Findutils*, and *Zlib* on the single representation model.

Opt	Target	Accuracy	Standard Deviation	Opt	Target	Accuracy	Standard Deviation	Opt	Target	Accuracy	Standard Deviation
Semantic	RepFTI_B	0.71	0.281	Graph	RepFTI_B	0.69	0.252	Semantic + Graph	RepFTI_B	0.85	0.059
	RepFTI_C	0.62	0.335		RepFTI_C	0.62	0.248		RepFTI_C	0.85	0.067
	RepFTI_F	0.79	0.258		RepFTI_F	0.60	0.254		RepFTI_F	0.89	0.039
	RepFTI_Z	0.83	0.191		RepFTI_Z	0.60	0.142		RepFTI_Z	0.87	0.061
	InnerEye	0.45	0.186		HGNN	0.20	0.148		Structure2Vec	0.19	0.098
	Asm2Vec	0.86	0.246		HGMN	0.64	0.265		Order matters	0.83	0.053
	EKLAVYA	0.83	0.226		GMNs	0.66	None		PalmTree	0.80	0.049

5. Discussion

It is worth noting that the proposed RepFTI utilizes a multi-representation fusion technology to reconstruct the whole data dependency of target functions to infer their types. We break down the limitations of independent representation by concatenating semantic-representation features and graph-representation features, which effectively adapts the environment of vehicular software systems. To evaluate the intra-functional semantic learning capacity, we establish a baseline by attempting to restore the features of the actual target function. We measure the feature vector of the filled execution path and the key reference feature vector. The results demonstrate the effective control of the cosine distance within 0.6, thereby increasing the accuracy of the downstream model evaluation for the pre-trained model. These results align with our expectations and indicate that the recovery of data dependence within the function meets the target requirements. To evaluate the recognition capability of the function-calling relationship, we analyze the structural characteristics of function calls for the target node based on the call structures between the target function node and its adjacent nodes across all execution paths. Using all nodes of the execution paths related to the target function under the condition of achieving 90% code coverage for the same software, we evaluate the data. We use a TSNE graph to visualize the clustering of the same function, and the final results demonstrate a good clustering effect for the same function across different execution paths. Since the RepFTI comprehensively uses MLM, Bi-LSTM, and GAT models, a new research route is exploited, which is scalable toward pre-trained language models. The experimental evaluation of the immutability of data-processing logic, the accuracy of RepFTI, and the reliability of RepFTI under different compilation optimization levels demonstrates the advantages of our methods.

6. Conclusions

In this paper, the immutability of data-processing logic under semantic presentation scenarios is validated for the cosine distance fluctuation that is less than 0.2. And then, by reducing high-dimensional data to two dimensions, we can observe that different functions are distinguishable on the TSNE plot, which is crucial to improving the security of complex vehicular software systems. Motivated by this, we propose the representation-fused function-type inference (RepFTI) to enhance the security of vehicular software systems. The RepFTI fuses semantic features within functions and call relationships of functions to repair long-span data dependencies of the target function. By comparing the true and false positive rates to existing methods, the effectiveness of RepFTI is demonstrated. To enable the real-time monitoring and analysis of software-execution states, we additionally employ a novel dynamic tracking strategy to simultaneously capture instruction sequences and function call graphs, obtaining the corresponding function type. Experimental results show that the RepFTI significantly enhances the accuracy and stability of function-type recognition compared to existing methods.

However, there is still room for improvement. The graph representation and semantic representation can be adaptively adjusted based on the analysis objective. Furthermore, annotating a large volume of data is a complex task, and improvements can be made in the annotation process for a wide range of function types.

Author Contributions: Conceptualization, X.Y., G.L., J.L. and A.D.; methodology, G.L. and J.L.; experiment and analysis, X.Y. and A.D.; writing——review and editing, X.Y. and G.L.; supervision, G.L. and J.L.; funding acquisition, G.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research work is funded by the National Nature Science Foundation of China under Grant No. U20B2048 and 62202303.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available upon request from the corresponding author. The data are not publicly available due to privacy and ethical concerns.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Yu, D.; Xiao, A. The Digital Foundation Platform—A Multi-layered SOA Architecture for Intelligent Connected Vehicle Operating System. *arXiv* **2022**, arXiv:2210.08818.
2. Cao, Y.; Xiao, C.; Cyr, B.; Zhou, Y.; Park, W.; Rampazzi, S.; Chen, Q.A.; Fu, K.; Mao, Z.M. Adversarial sensor attack on lidar-based perception in autonomous driving. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2267–2281.
3. Miller, C. Lessons learned from hacking a car. *IEEE Des. Test* **2019**, *36*, 7–9. [CrossRef]
4. Dibaei, M.; Zheng, X.; Jiang, K.; Abbas, R.; Liu, S.; Zhang, Y.; Xiang, Y.; Yu, S. Attacks and defences on intelligent connected vehicles: A survey. *Digit. Commun. Netw.* **2020**, *6*, 399–421. [CrossRef]
5. Hu, S.; Zhang, Q.; Weimerskirch, A.; Mao, Z.M. Gatekeeper: A gateway-based broadcast authentication protocol for the in-vehicle Ethernet. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May–3 June 2022; pp. 494–507.
6. Ebrahimi, M.; Marksteiner, S.; Ničković, D.; Bloem, R.; Schögler, D.; Eisner, P.; Sprung, S.; Schober, T.; Chlup, S.; Schmittner, C.; et al. A Systematic Approach to Automotive Security. In Proceedings of the International Symposium on Formal Methods, Lübeck, Germany, 6–10 March 2023; pp. 598–609.

7. Haney, J.M.; Lutters, W.G. "It's {Scary… It's}{Confusing… It's} Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security. In Proceedings of the Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018 Baltimore, MD, USA, 12–14 August 2018; pp. 411–425.
8. Jing, P.; Tang, Q.; Du, Y.; Xue, L.; Luo, X.; Wang, T.; Nie, S.; Wu, S. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 3237–3254.
9. Jing, P.; Cai, Z.; Cao, Y.; Yu, L.; Du, Y.; Zhang, W.; Qian, C.; Luo, X.; Nie, S.; Wu, S. Revisiting Automotive Attack Surfaces: A Practitioners' Perspective. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2024; pp. 80–80.
10. Saxena, P.; Poosankam, P.; McCamant, S.; Song, D. Loop-extended symbolic execution on binary programs. In Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, Chicago, IL, USA, 19–23 July 2009; pp. 225–236.
11. Song, D.; Brumley, D.; Yin, H.; Caballero, J.; Jager, I.; Kang, M.G.; Liang, Z.; Newsome, J.; Poosankam, P.; Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In Proceedings of the Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, 16–20 December 2008; pp. 1–25.
12. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Not.* **2011**, *46*, 265–278. [CrossRef]
13. Hemel, A.; Kalleberg, K.T.; Vermaas, R.; Dolstra, E. Finding software license violations through binary code clone detection. In Proceedings of the 8th Working Conference on Mining Software Repositories, Honolulu, HI, USA, 21–22 May 2011; pp. 63–72.
14. Sæbjørnsen, A.; Willcock, J.; Panas, T.; Quinlan, D.; Su, Z. Detecting code clones in binary executables. In Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, Chicago, IL, USA, 19–23 July 2009; pp. 117–128.
15. Ghormley, D.P.; Rodrigues, S.H.; Petrou, D.; Anderson, T.E. SLIC: An Extensibility System for Commodity Operating Systems. In Proceedings of the USENIX Annual Technical Conference, New Orleans, LA, USA, 15–19 June 1998; Volume 98.
16. Friedman, S.E.; Musliner, D.J. Automatically repairing stripped executables with cfg microsurgery. In Proceedings of the 2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, Cambridge, MA, USA, 21–25 September 2015; pp. 102–107.
17. Schulte, E.M.; Weimer, W.; Forrest, S. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, Madrid, Spain, 11–15 July 2015; pp. 847–854.
18. Christodorescu, M.; Jha, S.; Seshia, S.A.; Song, D.; Bryant, R.E. Semantics-aware malware detection. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland, CA, USA, 8–11 May 2005; pp. 32–46.
19. Kruegel, C.; Robertson, W.; Vigna, G. Detecting kernel-level rootkits through binary analysis. In Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, USA, 6–10 December 2004; pp. 91–100.
20. Kruegel, C.; Kirda, E.; Mutz, D.; Robertson, W.; Vigna, G. Automating mimicry attacks using static binary analysis. In Proceedings of the USENIX Security Symposium, Baltimore, MD, USA, 1–5 August 2005; Volume 14, pp. 161–176.
21. Shin, E.C.R.; Song, D.; Moazzezi, R. Recognizing functions in binaries with neural networks. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 611–626.
22. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the IEEE S&P, San Francisco, CA, USA, 20–22 May 2019; pp. 472–489.
23. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; pp. 58–79.
24. Guo, W.; Mu, D.; Xing, X.; Du, M.; Song, D. {DEEPVSA}: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1787–1804.
25. Jin, X.; Pei, K.; Won, J.Y.; Lin, Z. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 1631–1645.
26. Sun, X.; Wei, Q.; Du, J.; Wang, Y. HEBCS: A High-Efficiency Binary Code Search Method. *Electronics* **2023**, *12*, 3464. [CrossRef]
27. Liu, S. A unified framework to learn program semantics with graph neural networks. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 December 2020; pp. 1364–1366.
28. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376.
29. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In Proceedings of the International Conference on Machine Learning, PMLR, Long Beach, CA, USA, 9–15 June 2019; pp. 3835–3845.
30. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
31. Zuo, F.; Li, X.; Young, P.; Luo, L.; Zeng, Q.; Zhang, Z. Neural machine translation inspired binary code similarity comparison beyond function pairs. In Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS, San Diego, CA, USA, 24–27 February 2019.

32. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 896–899.
33. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. In Proceedings of the AAAI, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1145–1152.
34. Li, X.; Qu, Y.; Yin, H. Palmtree: Learning an assembly language model for instruction embedding. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Virtual, 15–19 November 2021; pp. 3236–3251.
35. Xue, Y.; Xu, Z.; Chandramohan, M.; Liu, Y. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Trans. Softw. Eng.* **2018**, *45*, 1125–1149. [CrossRef]
36. Perkins, J.H.; Kim, S.; Larsen, S.; Amarasinghe, S.; Bachrach, J.; Carbin, M.; Pacheco, C.; Sherwood, F.; Sidiroglou, S.; Sullivan, G.; et al. Automatically patching errors in deployed software. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 87–102.
37. Van Der Veen, V.; Göktas, E.; Contag, M.; Pawoloski, A.; Chen, X.; Rawat, S.; Bos, H.; Holz, T.; Athanasopoulos, E.; Giuffrida, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 934–953.
38. Lee, J.; Avgerinos, T.; Brumley, D. TIE: Principled reverse engineering of types in binary programs 2011. In Proceedings of the NDSS on Network and Distributed System Security Symposium, San Diego, California, USA, 6 February–9 February 2011. Available online: https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs (accessed on 5 July 2021).
39. Chua, Z.L.; Shen, S.; Saxena, P.; Liang, Z. Neural nets can learn function type signatures from binaries. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 99–116.
40. intel. Pintools. 2007. Available online: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (accessed on 5 July 2021).
41. intel. RTN. 2007. Available online: https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__RTN.html (accessed on 5 July 2021).
42. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
43. Yi, X.; Wu, J.; Li, G.; Bashir, A.K.; Li, J.; AlZubi, A.A. Recurrent Semantic Learning-driven Fast Binary Vulnerability Detection in Healthcare Cyber Physical Systems. *IEEE Trans. Netw. Sci. Eng.* **2022**, *10*, 2537–2550. [CrossRef]
44. Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph attention networks. *arXiv* **2017**, arXiv:1710.10903.
45. Reddivari, S.; Wolbert, J. Calculating Requirements Similarity Using Word Embeddings. In Proceedings of the IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), Los Alamitos, CA, USA, 27 June–1 July 2022; pp. 438–439.
46. Ling, X.; Wu, L.; Wang, S.; Ma, T.; Xu, F.; Wu, C.; Ji, S. Hierarchical graph matching networks for deep graph similarity learning. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '21), Singapore, 14–18 August 2021; Association for Computing Machinery, New York, NY, USA, 2021; pp. 2274–2284. [CrossRef]
47. Liu, S.; Chen, Y.; Xie, X.; Siow, J.; Liu, Y. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv* **2020**, arXiv:2006.05405.