CAPS: An Efficient Whole-Program Critical Paths Search Framework for Large-Scale Software

Peiyang Li, Zixin Liu, Yuening Su, Hao Wang, Bo Jiang*
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University, China
{lipeiyang, liuzixin, Su_Yuening, wangritian, jiangbo}@buaa.edu.cn

Abstract—**Tracking the flow of external inputs in a program with taint-analysis techniques can help developers better identify potential security vulnerabilities in the software. However, directly using the static taint analysis provided by Clang Static Analyzer is inefficient for large-scale software due to the huge but redundant ExplodedGraph generated. Therefore, we propose an efficient Whole-Program Critical Paths Search (CAPS) framework. It first performs a set of optimizations to reduce the ExplodedGraph of each function. Then, it constructs a global exploded graph by inserting call edges among the reduced ExplodedGraphs for each function within the Neo4j graph database. Finally, it proposes loop removal and graph segmentation to optimize the search process for critical paths on the global exploded graph. Our experiments on 3 large-scale software show that CAPS can significantly improve the efficiency of critical path search for large-scale software.**

Keywords-component; critical path search; static analysis; Clang Static Analyzer; ExplodedGraph; taint analysis

I. INTRODUCTION

For modern software systems, external input data has the potential to trigger its underlying vulnerabilities [1]. Many taint analysis techniques [2][3][4] have been proposed to identify these vulnerabilities or to detect privacy leaks. These techniques typically start by obtaining the control flow graph and data flow graph of the program. Then, external inputs (e.g., input parameters) or other untrusted data are marked as sources on the data flow graph. A taint analysis is performed on the graph to track the impact of the sources within the program until the sinks are reached. A typical taint analysis requires the sinks to be defined and identified in advance. Taint tracking cannot be performed without the knowledge of the sinks. However, defining and identifying sinks for large-scale complex software can be difficult. Furthermore, in our work, we are only interested in identifying all the paths reachable from a source to guide manual security check or fuzzing. Therefore, our goal in this work is to perform whole-program critical paths search for a large software. In this work, the **whole-program critical paths** are defined as all the paths reachable from each public interface function through static taint analysis by marking each of its parameter as tainted.

Many program static analysis techniques have been proposed for program path analysis, such as Clang Static Analyzer (CSA)[5], SVF[12], Phasar [13], etc. These techniques typically use symbolic execution or data flow analysis methods to explore executable paths in a program. Specifically, CSA is a source code analysis tool that represents all input values (e.g., function parameters) as symbolic values and performs path-sensitive code analysis using symbolic execution techniques. Given the entry function of a software, it explores all possible execution paths in the whole program and calculate the symbolic values of expressions in the program, such that all expressions related to input values are represented as a function of the input symbolic values. The set of explored paths is represented using an ExplodedGraph. However, performing taint analysis by feeding the entry function to Clang Static Analyzer cannot track all the critical paths starting from each public interface function. Furthermore, if we perform inter-procedural taint analysis on each public interface function, many functions will be repeatedly analyzed, leading to explosive growth in analysis time.

To address the above problem, we adopt CSA to perform intra-procedural analysis on all functions to obtain their respective ExplodedGraphs. Next, we extract the entities and entity relationships from each ExplodedGraph and add the call relationships between functions. Then, we import these entities and relationships into the Neo4j [6] graph database to build the global exploded graph for the software. Finally, we perform search on the global exploded graph within the graph database to find all critical paths for an interface function.

However, we found the above approach to find critical paths within the global exploded graph is inefficient due to two reasons. First, the ExplodedGraph contains a large number of redundant nodes, which makes the search space too large to search efficiently. To solve this problem, we propose a technique to reduce the scale of the ExplodedGraph by merging and deleting nodes that are irrelevant to the tainted function parameters. Our experiment shows the technique can reduce the scale of the original ExplodedGraph by approximately 80%. Second, when searching for the critical path on the global exploded graph, we found that there are some loops and repeated paths, which further reduce the efficiency of path search. To solve this problem, we propose an optimization algorithm for path search that can significantly improve its efficiency. Based on the above proposals, we build a whole-program Critical Paths Search (CAPS) framework for efficient critical path search of large-scale software.

The contributions of this paper are as follows:

- We build a custom Checker based on CSA, called ExplodedGraphEmitChecker. The Checker generates a

DOI reference number: 10.18293/SEKE2023-123

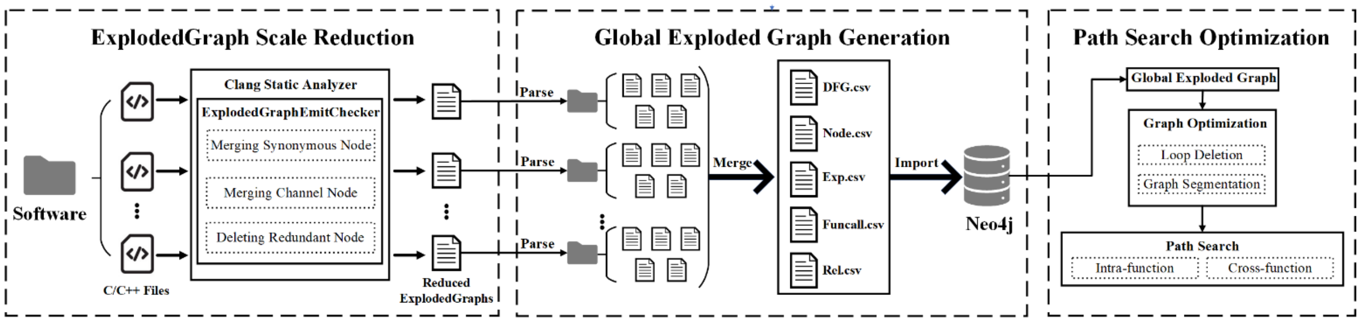

Figure 1. The Whole-Program Critical Paths Search (CAPS) Framework.

reduced ExplodedGraph for each functional unit by merging and removing nodes that are not related to the critical path in the original ExplodedGraph.

- We extract entities and their relationships from the reduced ExplodedGraph of each function and add function call relationships, and then use the Neo4j graph database to build the global exploded graph for large software.
- We propose an optimized whole-program critical paths search method based on the depth-first search algorithm. This method reduces the search space through loop deletion and graph segmentation, thereby improving the efficiency for critical path search.
- We have systematically evaluated our framework on 3 large GNU software. The experimental results show that it can significantly improve the search efficiency of critical paths on the global exploded graph.

The remaining sections are organized as follows. In Section II we detail our proposed whole-program critical paths search (CAPS) framework for large software. In Section III we conduct experiments with CAPS on 3 large GNU software and discuss the experimental results. Sections IV and V present related work and conclusions.

II. WHOLE-PROGRAM CRITICAL PATHS SEARCH

In this section, we present the Whole-Program Critical Paths Search (CAPS) framework as shown in Fig. 1. The framework is mainly divided into three modules: ExplodedGraph scale reduction, global exploded graph generation, and path search optimization. Next we will describe each module in detail.

A. *ExplodedGraph Scale Reduction*

The scale of the original ExplodedGraph is large. Because CSA's symbolic execution engine generate multiple ExplodedNodes containing the program state and program points for each analyzed statement at the corresponding program location, which can easily cause the rapid growth of the number of nodes. By analyzing these nodes, it can be discovered that there are three types of ExplodedNodes in the original ExplodedGraph that are irrelated to the critical path: a) **Synonymous nodes**. These nodes are adjacent in location and have the same state information. b) **Channel nodes**. These nodes have a unique predecessor node and successor node, and do not involve branching and merging of paths. c) **Redundant nodes**. These nodes are not related with program input values and do not exist on the critical path. In this module, we generate a scale-reduced ExplodedGraph for each functional unit by processing the above three types of nodes.

Merging Synonymous Node. If there is a subgraph G_s in the original ExplodedGraph G where all nodes in G_s have exactly the same expression information and program state information, then all nodes in G_s are merged into a single node n_s. The merged node n_s inherits all the previous predecessor and successor relationships between internal nodes and external nodes in G_s.

Merging Channel Node. If there is a subgraph G_c in the original ExplodedGraph G where all nodes are channel nodes (i.e., each node only has one predecessor and one successor), then all nodes in G_c are merged into a single node n_c. The merged node n_c contains all the information of all nodes in G_c and also retains all the previous predecessor and successor relationships between internal nodes and external nodes in G_c.

Deleting Redundant Node. If there is a node n_r in the ExplodedGraph, and the ProgramState of n_r does not contain any symbolic information about function parameters, then n_r will be deleted, and the original predecessor node and successor node of n_r will be adjacent to each other.

Based on the above method, we reduced the number of nodes in the original ExplodedGraph, thereby reducing the scale of the ExplodedGraph and improving the efficiency of the subsequent critical path search. It should be noted that the above method is proposed under the premise of satisfying the following two principles:

- Graph isomorphism. When all nodes in a subgraph G' are merged into a single node n, if an external node of the G' is a predecessor node of a node in G', then it is also a predecessor node of node n; if an external node of the G' is a successor node of a node in G', then it is also a successor node of node n.
- Information consistency. If the state information and function call information in the node exists in the original ExplodedGraph, it still exists in the reduced ExplodedGraph.

Graph isomorphism ensures the synonymity of paths in the reduced ExplodedGraph. That is to say, any path passing through any node in subgraph G' in the original ExplodedGraph will also pass through the merged node n in the reduced ExplodedGraph. Information consistency ensures that any path search strategy on the original ExplodedGraph and the reduced ExplodedGraph before and after reduction will yield consistent results.

B. *Global Exploded Graph Generation*

We use the graph database Neo4j to build a global exploded graph for large-scale software. In this graph, each node represents an entity in the ExplodedGraph (such as a function, ExplodedNode, expression, or function call point), and each entity has multiple attribute information as shown in TABLE I. Edges represent the relationships between these entities. For example, since an ExplodedGraph corresponding to a function contains multiple ExplodedNodes, we use INCLUDE to represent the relationship between the function and the ExplodedNode, as shown in TABLE II. We extract entities and relationships between them by parsing the reduced ExplodedGraph corresponding to each functional unit. However, there are many functions in a large software, meaning that there are multiple ExplodedGraphs. In order to build the global exploded graph, we merge the parsing results corresponding to all the ExplodedGraphs and import them into Neo4j in batch. In the global exploded graph, if there is a calling relationship between functions, we will connect the corresponding reduced ExplodedGraphs of the functions using the FunCall relationship in TABLE II.

TABLE I. ENTITIES IN EXPLODEDGRAPH

Entity Type	Entity Attribute Set	Actual Object
DFG	id, line, moduleName, parameters, szFun, szOrg	Function
Node	id, line, moduleName, szFun, szpretty	Simplified ExplodedNode
Exp	id, szExpName, szExpValue	Expression
FunCall	id, moduleName, szFileOfFuncFirstDecl, szFun, szPrototype, szRareStmt	Function call point

TABLE II. RELATIONSHIP BETWEEN ENTITIES

Relationship	Start	End	Quantity ratio
ENTRY	DFG	Node	1:1
INCLUDE	DFG	Node	1:n
NEXT	Node	Node	n:m
EXP	Node	Exp	1:n
CALL	Node	FunCall	1:1
CALLEE	FunCall	DFG	1:1

In TABLE I. , line represents the actual location of the entity in the source code; moduleName represents the project filename that the entity belongs to; szFun represents the function that the entity belongs to and its parameters; szOrg represents the absolute path of the project file that the entity belongs to; szpretty represents the source code statement; szExpName represents the source expression; szExpValue represents the symbol value corresponding to the source expression; szFileOfFuncFirstDecl represents the file address where the called function is located; szPrototype represents the name and parameters of the called function; szRareStmt represents the function call statement. In TABLE II. , ENTRY represents the relationship between a function and its entry node, where each function has only one entry; INCLUDE represents the relationship between a function and its ordinary nodes; NEXT represents the sequential relationship between nodes; EXP represents the relationship between a node and multiple expressions within the node; CALL represents the relationship between a node and its function call points; CALLEE represents the relationship between a function call point and the called function.

C. *Path Search Optimization*

The complexity of the original ExplodedGraph structure greatly affects the efficiency of path search on the global exploded graph. Because CSA caches and reuses some of the same ExplodedNodes in order to improve analysis performance and avoid repeated calculations during the process of generating the ExplodedGraph of the functional unit. However, this caching and reuse mechanism may lead to a reverse path from the current node to the previous node, forming a loop, which indirectly leads to an infinite loop of the path search algorithm. In addition, the number of paths of ExplodedGraph corresponding to complex functions is usually exponential, which greatly affects the search efficiency. Therefore, in order to improve the search efficiency of the critical path on the global exploded graph, we propose the CAPS framework. CAPS first optimizes the intra-function path search method based on the naive DFS (Depth First Search) algorithm. The optimizations include loop removal and graph segmentation. After the search for intra-function paths is completed, CAPS combines multiple intra-function paths to generate a complete cross-function path based on the global exploded graph, achieving whole-program tracing of the parameters.

The task of searching intra-function paths can be described as *IntraPathSet = searchIntraPath* (*Func*, *SelectParamSet*). Its inputs are a function (*Func*) and a set of parameter symbol values (*SelectParamSet*), and its output is a set of parameter-related intra-function paths. An intra-function path refers to a path that satisfies the following conditions:

- The path is an ordered list of nodes in the ExplodedGraph of a function (*Func*).
- The starting node of the path is the unique entry node of the ExplodedGraph.
- The ending node of a path is the target node in the ExplodedGraph, i.e., the node that contains one of the parameters in *SelectParamSet*.

1) *Graph Optimization*

Before searching intra-function paths on ExplodedGraph, CAPS sequentially performs the following optimization processes: loop deletion and graph segmentation.

a) Loop Deletion: The loop deletion transforms the ExplodedGraph into a Directed Acyclic Graph (DAG) by deleting some edges from the original ExplodedGraph. Specifically, the CAPS first performs a DFS traversal on the original ExplodedGraph starting from the entry node, to define the priority of all nodes. The later a node is traversed by the DFS, the higher its priority is. Then CAPS deletes all edges that satisfy the following condition (i.e., loop edges): *Priority(st)* $\leq$ *Priority(en)*, where *st* and *en* are respectively the starting and ending nodes of the edge.

b) Graph Segmentation: Graph segmentation is another method to solve the problem of exponential intra-function path number. It makes use of redundant information among intra-function paths and changes the path's data representation. The purpose of graph segmentation is to divide the nodes of a DAG graph into two sets S1 and S2, which are close in size (Fig. 2). Graph segmentation satisfies the following principles:

- Any intra-function path ending at S1 node (a node in set S1) contains no S2 node.
- Any intra-function path ending at S2 node contains a special S1 node called *barrierNode* (Barrier Node), which can divide this path into two parts: the former part is a path contains no S2 node (An intra-function path ending at *barrierNode*), the latter part is a path contains no S1 node.

In this way, all intra-function paths that have the same *barrierNode* can be jointly represented as <*FormerPathSet*> ⊗ <*LatterPathSet*>. *FormerPathSet* is a set of several intra-function paths ending at *barrierNode*. *LatterPathSet* is a set of several paths contain no S1 node. The symbol ⊗ represents the Cartesian product of two sets. This path representation can save storage space, and limits the DFS path search range to set S1 or S2 (instead of the entire ExplodedGraph). As shown in Fig. 2.

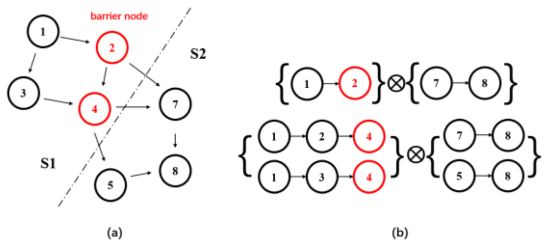

Figure 2. (a) The node set S1，S2 and barrier node in graph segmentation. (b) All the paths represented by symbol ⊗ from node 1 to node 8.

2) *Search of Intra-function and Cross-function Path*

The task of intra-function path search can be described as *IntraPathSet* = *searchIntraPath* (*Func*, *SelectParamSet*). Because of the existence of graph segmentation algorithm, the elements of *IntraPathSet* have the form *innerpath* = <*formerpathset*> ⊗ <*Latterpathset*>. If the ending node of an intra-function path has function call, then the path can be extended to other functions. In the process of cross-function path search, the procedure of *searchIntraPath*(*Func*, *SelectParamSet*) is called continuously, and several intra-function paths are combined into a complete cross-function path.

D. Limitation

However, we found that the critical path search method, which searches based on the symbol value of function parameters, is limited by the analysis ability of the CSA itself, and sometimes the complete path cannot be tracked. The specific reason is that CSA is currently unable to accurately infer the symbol values of some expressions (e.g., floating-point variables, complex type variables, function return values, etc.), which causes the interruption of the propagation of parameter symbol values.

III. Experiment and Evaluation

In this section, we conduct experiments on 3 large-scale open-source GNU software and evaluate our CAPS.

A. Reasearch Questions

RQ1: Can CAPS effectively reduce the scale of the ExplodedGraph?

RQ2: Can CAPS effectively improve the efficiency of critical path search for large-scale software?

B. Experiment Design

1) *Subject Programs*

We selected three real large-scale GNU software to evaluate CAPS, as shown in TABLE III. Tar [7] is a widely used archiving and packaging software on UNIX and UNIX-like systems, which can combine multiple files into a single file. In addition, Tar also has other file operation functions, such as extraction, storage, etc. Mailutils [8] is a protocol-independent framework for email processing. It provides a set of libraries for doing almost any mail-related task on any existing mailbox format, using a consistent format-independent API. M4 [9] is a macro processor in the sense that it copies its input to the output expanding macros as it goes. Besides just doing macro expansion, M4 has built-in functions for including named files, running UNIX commands, doing integer arithmetic, manipulating text in various ways, recursion etc.

TABLE III. Projects Used in Our Experiment

Subject	Version	Loc
Tar	1.33	102k
Mailutils	3.13	209k
M4	1.4.19	142k

2) *Experiment Setup*

We performed our experiments on a desktop with Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz and 32GB of memory. The operating system is Ubuntu 20.04 LTS. The version number of Clang used to generate the ExplodedGraph for the functional unit is 15.0.0. The version number of the graph database Neo4j used to build the global exploded graph for the software is 4.3.12.

3) *Experiment Procedure*

For each open-source software, we first use CAPS to generate a reduced global exploded graph, and then compare it with the unreduced global exploded graph in terms of node and relationship counts to verify the effectiveness of CAPS in achieving ExplodedGraph scale reduction. Next, we use the naive depth-first search algorithm as a baseline to verify the performance advantages of the path search optimization method in CAPS on the reduced global exploded graph.

We define metrics Path Search Rate (PSR) and Weighted Rate Ratio (WRR) to evaluate the search efficiency of search algorithms in critical path search. PSR is the ratio of the number of critical paths found to the search time when searching for the critical path of a parameter of a function in the global exploded graph. Because it is impossible to calculate the PSR of all function parameters within a limited amount of time in a large software with numerous functions and parameters. Therefore, in order to effectively evaluate the performance of search algorithms in the global exploded graph, we first randomly select 200 functions parameters from the software for critical path search, and choose the parameters with path counts in the top 25% as the experimental validation set for calculating their PSR metrics. We chose parameters that involve more critical path numbers as our validation set because functions with fewer critical paths have very small differences in PSR between CAPS and DFS, which can be almost negligible. Then, we compare the performance of CAPS and naive DFS algorithms according to (1), where $PSR_{CAPS}(i)$ and $PSR_{DFS}(i)$ respectively represent the search rate of these two methods when performing critical path search on the *i* th function parameters.

$$Weighted\ Rate\ Ratio(WRR) = \frac{\sum_{i=1}^{n} \frac{PSR_{CAPS}(i)}{PSR_{DFS}(i)}}{n} \qquad (1)$$

C. Result and Analysis

In this section, we present our experimental results and analyze the research questions proposed in III(A).

1) Answering RQ1

TABLE IV. presents the comparison results of the number of nodes and edges in the global exploded graph of 3 GNU software before and after reduction. The “original” and “reduced” respectively represent the global exploded graph before and after reduction. The “reduction rate” shows the percentage decrease in the number of nodes or edges in the reduced graph compared to the original one. According to TABLE IV. , we can see that CAPS has the best performance in reducing the global exploded graph of M4, and its reduction rate of node and edge counts is 85.14% and 83.47%. Especially the reduction rate of the number of edges is 6.45% and 8.61% higher than that of Tar and Mailutils, respectively. Regarding the reduction of nodes, CAPS shows comparable performance with Tar and Mailutils, achieving reduction of 80.58% and 80.17%. Overall, the average reduction rate of nodes and edges is 81.96% and 78.45% respectively, which indicates the effectiveness of CAPS in ExplodedGraph scale reduction.

TABLE IV. COMPARISON OF THE NUMBER OF NODES AND EDGES BEFORE AND AFTER GLOBAL EXPLODED GRAPH REDUCTION

Project		Node	Edge
Tar	original	20,026,774	138,976,378
	reduced	3,889,400	31,938,078
	reduction rate	**80.58%**	**77.02%**
Mailutils	original	25,972,727	186,716,558
	reduced	5,149,298	46,940,666
	reduction rate	**80.17%**	**74.86%**
M4	original	12,027,333	86,891,874
	reduced	1,787,609	14,360,459
	reduction rate	**85.14%**	**83.47%**

Furthermore, from the reduction rate of the number of nodes, we can further analyze that most of the nodes in the original ExplodedGraph are irrelevant to function parameters. Therefore, by optimizing and removing these nodes, we can greatly improve the efficiency of subsequent critical path search.

2) Answering RQ2

TABLE V. shows the weighted rate ratio (WRR) of the 3 GNU software after critical path search using CAPS and naive DFS algorithms at five different search time of 30s, 60s, 90s, 120s and 150s. Larger WRR indicates that CAPS is more efficient than naive DFS in critical path search. According to TABLE V. , we can see that for Mailutils, the average WRR is the highest, reaching 41.56. On the other hand, for Tar, the average WRR is lowest, only 7.39. Furthermore, for the same project, the WRR does not increase linearly with the increase in search time. These indicate that the performance advantage of CAPS over the naive DFS algorithm may be affected by the intrinsic structure of the software.

When we observe the WRR under different software and different search time, we can find that their values are all larger than 1, and the minimum value is 1.82, which shows that CAPS outperforms the naive DFS algorithms in critical path search efficiency, regardless of the software and the search time. From the above experimental results, we conclude that the CAPS is capable of effectively improving the search efficiency of the critical path for large-scale software.

TABLE V. COMPARISON OF WRR AT DIFFERENT SEARCH TIME

Project	Weighted Rate Ratio					Avg
	30s	60s	90s	120s	150s	
Tar	4.16	1.91	1.82	25.46	3.6	**7.39**
Mailutils	40.18	7.3	34.34	37.64	88.35	**41.56**
M4	21.86	18.99	12.14	19.49	56.53	**25.80**

IV. RELATED WORK

In this section, we review related works on taint analysis and global exploded graph generation.

A. Taint analysis

Taint analysis techniques [10][11] are commonly used to identify information flows in programs, tracking the movement of sensitive information from a set of sensitive sources to sensitive sinks. Sui et al. [12] have developed a static analysis tool called SVF, which integrates the functionalities of pointer analysis, value flow analysis, and taint analysis. The tool takes LLVM IR (Intermediate Representation) as input and uses analysis modules such as Program Assignment Graph (PAG) and Control Flow Graph (CFG) provided by LLVM to perform whole-program pointer analysis, draw Sparse Value-Flow Graph (SVFG) value flow graph, and ultimately obtain the flow representation of each data element within the program. Phasar [13] is a large-scale C/C++ program static analysis framework developed by Philipp et al. It performs data flow and control flow analysis based on the LLVM-IR of the input program to obtain the Interprocedural Control Flow Graph (ICFG) data flow graph. Then, the framework uses the Interprocedural Finite Distributive Subset (IFDS) algorithm to achieve complete taint analysis. Additionally, Phasar also includes a simple Boomerang pointer analysis tool, enabling it to detect some synonym pointer propagation during program execution and fill in the missing parts of the IFDS algorithm analysis. She et al. [14] proposed a novel end-to-end method to track information flow by using neural network, called Neutaint. It models target program computations that occur between taint sources and sinks, and automatically learns information flow by observing a set of different execution traces. Experimental results show that Neutaint can achieve an average accuracy rate of 68%. Zhang et al. [15] developed FastDroid, a tool for detecting sensitive data leaks in Android applications. It first constructs a taint value graph (TVG) by flow-insensitive taint analysis to describe the taint propagation process. Then, potential taint flows are extracted from TVG. Finally, it compares the potential taint flows with the control flow graph to obtain the real taint flows. The results show that FastDroid can improve the analysis efficiency while ensuring high precision and recall. However, compared with CAPS, most of the above methods are only suitable for local program analysis, and cannot perform whole program critical path search and taint analysis.

B. *Global exploded Graph generation*

The global exploded graph [16] can describe program paths within functions as well as between functions through edges. Building the global exploded graph can help analysis tools more accurately understand program behavior and achieve global static analysis [17]. Gharibi et al. [18] developed a program analysis tool called code2graph that can automatically analyze source code, construct its static call graph, generate all possible execution paths of the system, and calculate their similarities. Abdelaziz et al. [19] designed a toolkit for building code knowledge graphs called GraphGen4Code. GraphGen4Code uses generic techniques to capture code semantics, and key nodes in the graph represent classes, functions, and methods. Edges represent the call relationship between functions. It can serve applications such as program search, code understanding, error detection, and code automation. However, the program graph constructed by the above methods based on the control flow graph and call graph, and its granularity is relatively coarse. In contrast, the global exploded graph constructed by CAPS is composed of program point and program state, and the granularity is smaller, thus providing more accurate analysis.

V. Conclusion

Many taint analysis techniques have been proposed to track the flow of external inputs in the program, so as to identify potential security vulnerabilities in the software. However, these techniques require the sinks to be defined and identified in advance, which is difficult for large-scale software. If static taint analysis is performed directly on the original ExplodedGraph generated by Clang Static Analyzer, although the sinks does not need to be identified, there are many nodes in the original ExplodedGraph that are irrelevant to external input, which makes the search of the critical path inefficient. Therefore, we propose an efficient Whole-Program Critical Paths Search (CAPS) framework. The framework first implements ExplodedGraph reduction corresponding to each function through node merging and deleting. Then, it utilizes the entities and relationships existing in the reduced ExplodedGraph of each function and the calling relationship between functions, to construct a global exploded graph for large-scale software within Neo4j graph database. Finally, it optimizes the critical path search process by loop removal and graph segmentation. Our experimental results on 3 large GNU software demonstrate that CAPS can significantly reduce the original ExplodedGraph scale and improve the efficiency of critical path search on the global exploded graph for large-scale software.

For future work, we plan to explore more efficient algorithms and optimization techniques to improve the efficiency of whole-program critical paths search. Furthermore, we will attempt to resolve the breakpoint issue currently encountered during critical path searching to enhance the completeness of the path.

References

[1] A. M. Alashjee, S. Duraibi, J. J. I. J. o. C. S. Song, and Security, “Dynamic Taint Analysis Tools: A Review,” vol. 13, no. 6, pp. 231-244, 2019.

[2] J. Zhang, Y. Wang, L. Qiu and J. Rubin, “Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe,” in IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 4014-4040, 1 Oct. 2022.

[3] A. Davanian, Z. Qi, Y. Qu, and H. Yin, “DECAF++: Elastic Whole-System Dynamic Taint Analysis,” in RAID, 2019, pp. 31-45.

[4] J. Liang et al., “PATA: Fuzzing with path aware taint analysis,” in 2022 IEEE Symposium on Security and Privacy (SP), 2022, pp. 1-17: IEEE.

[5] Clang Static Analyzer. https://clang-analyzer.llvm.org/. Last access, 2023.

[6] Neo4j. https://neo4j.com/. Last access, 2023.

[7] Tar. https://www.gnu.org/software/tar/. Last access, 2023.

[8] Mailutils. https://mailutils.org/. Last access, 2023.

[9] M4. https://www.gnu.org/software/m4/. Last access, 2023.

[10] J. Zhang, Y. Wang, L. Qiu, and J. J. I. T. o. S. E. Rubin, “Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe,” vol. 48, no. 10, pp. 4014-4040, 2021.

[11] S. Arzt et al., “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” vol. 49, no. 6, pp. 259-269, 2014.

[12] Y. Sui and J. Xue, “SVF: interprocedural static value-flow analysis in LLVM,” in Proceedings of the 25th international conference on compiler construction, 2016, pp. 265-266.

[13] P. D. Schubert, B. Hermann, and E. Bodden, “Phasar: An inter-procedural static analysis framework for c/c++,” in Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II 25, 2019, pp. 393-410: Springer.

[14] D. She, Y. Chen, A. Shah, B. Ray and S. Jana, “Neutaint: Efficient Dynamic Taint Analysis with Neural Networks,” 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020, pp. 1527-1543.

[15] J. Zhang, C. Tian and Z. Duan, “FastDroid: Efficient Taint Analysis for Android Applications,” 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 2019, pp. 236-237

[16] H. J. U. d. S. Theiling, Diss, “Control flow graphs for real-time systems analysis,” 2002.

[17] W. Jia, Y. Wang, Y. Zhang, and Y. Gong, “Whole program paths generation method,” in 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS), 2018, pp. 1-4: IEEE.

[18] G. Gharibi, R. Tripathi, and Y. Lee, “Code2graph: automatic generation of static call graphs for python source code,” in Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 880-883.

[19] I. Abdelaziz, J. Dolby, J. McCusker, and K. Srinivas, “A toolkit for generating code knowledge graphs,” in Proceedings of the 11th on knowledge capture conference, 2021, pp. 137-144.