

RustPruner: A Program Slicing Tool for Rust Programs

Yanfeng Hu, Weihong Chen, Yilong Zhao, Ruiyu Zhang, Liangze Yin*, Wei Dong
College of Computer, National University of Defense Technology, Changsha, China
{huyanfeng22, chenweihong, zylong, zhangruiyu, yinliangze, wdong}@nudt.edu.cn

Abstract—In the realm of analyzing Rust programs, traditional full-scale analytical approaches are often rendered impractical due to considerable time and performance expenditures. Despite the potential for program slicing technologies to drastically curtail these expenses, the vast majority of existing slicing tools lack compatibility with the Rust language. This study introduces RustPruner, a specialized tool designed for slicing Rust code. RustPruner commences by constructing a goto-style control flow graph (CFG) through rigorous control flow analysis, comprehensively mapping out all feasible execution paths. Subsequently, it leverages a backward slicing algorithm, integrating the outcomes of program dependency analysis and abstract syntax tree (AST) analysis, to generate precise program slices. By innovatively adapting slicing technology to Rust, RustPruner facilitates comprehensive, low-level analysis of Rust’s distinctive features. The generated slices undergo rigorous compilation and verification procedures, thereby enhancing the efficiency of the analysis process. The effectiveness of RustPruner has been validated within some crucial system modules of operational operating systems, which significantly improves both the efficiency and accuracy of the verification.

Index Terms—Rust analysis, program slicing, control flow graph, abstract syntax tree

I. INTRODUCTION

Rust has rapidly emerged in the field of software development, championed by its safety features. Following its initial release in 2010, Rust has been extensively adopted across various critical domains. To guarantee the security and correctness of these systems, thorough code analysis and verification are required, which can significantly deplete time and performance resources. Program slicing can mitigate much of such resource expenditure by eliminating unnecessary analysis, making it especially vital in Rust development. However, despite the swift growth of Rust, there is a notable lack of slicing tools tailored for Rust.

Program slicing [1] is a method of code analysis that enables developers to extract code segments related to specific functionalities or requirements. Although Rust slicing holds significant potential in theory, it faces considerable challenges in practical application. Firstly, the unique ownership [2] and lifetime [3] rules of the Rust language complicate the application of slicing techniques. Secondly, the current lack of slicing tools tailored for the Rust language often necessitates reliance on manual analysis for Rust program slicing operations, which

is not only inefficient but also fails to fully leverage the distinctive features of the Rust language.

To address these issues, this paper presents RustPruner, a tool designed to provide Rust developers with an efficient and precise solution for code slicing. RustPruner resolves compatibility issues with traditional slicing tools by transforming programs into an intermediate goto representation that captures the language-specific features of Rust. Through backward analysis [4], it traces from the program’s outputs or assertions to all code that may affect the results, revealing the dependencies among various parts of the program. This analysis is especially crucial for understanding the system structure in large software projects. Through this process, we can obtain the most minimal code segments closely related to the target, effectively reducing interference from irrelevant code and enhancing the focus and accuracy of the analysis.

In the experimental section, this paper employs validation projects from real-world operating systems to assess the effectiveness and performance of RustPruner. The four projects in the experimental section all originate from the process scheduling module, which is a key module of the operating system. The experimental results indicate that RustPruner not only significantly reduces the amount of code but also maintains the consistency of verification properties through program verification operations on the generated slices. This enhances the efficiency and precision of verification and analysis tasks and also confirms the practicality of the tool in managing larger projects.

The contributions of this paper are as follows:

- 1) We propose a framework for program slicing in Rust that effectively tackles the challenges associated with Rust language slicing, thereby enhancing the efficiency and accuracy of code analysis.
- 2) Compatibility issues of Rust language features in traditional slicing tools are resolved, and efficient filtering and reorganization of code are achieved through precise AST analysis using a custom visitor.
- 3) The RustPruner tool is successfully implemented based on the proposed framework and its performance are evaluated in complex verification projects, demonstrating the tool’s validity and capabilities.

The rest of this paper is organized as follows. Section 2 provides background information. Section 3 discusses the technical framework of the tool. Section 4 evaluates the efficacy of RustPruner. Section 5 lists related literature.

This work was funded by the National Nature Science Foundation of China (No. U2341212, No. 62032024).

*Corresponding author

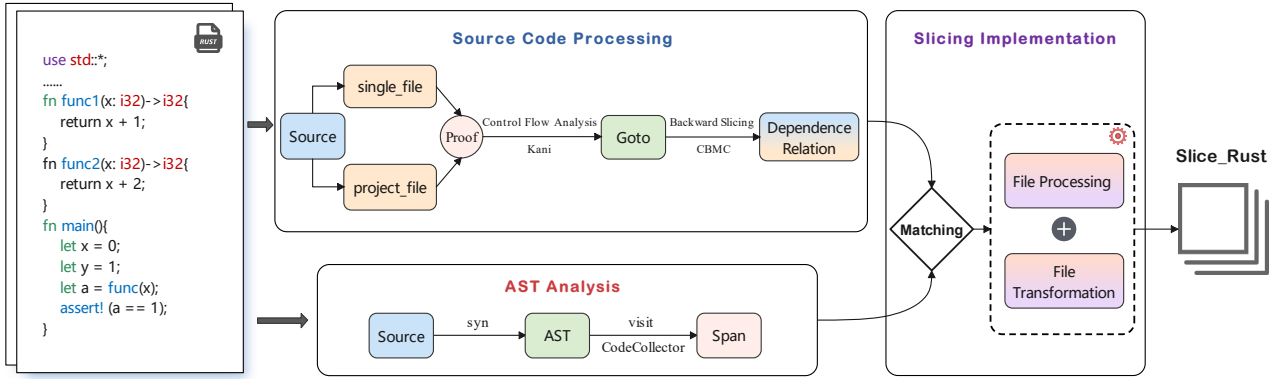


Fig. 1: The RustPruner Overview

II. BACKGROUND

A. Program Slicing

Program slicing [1] is a code analysis technique that extracts the smallest segment of code directly related to a specific goal from complex program code. This concept was first formally introduced by Weiser in his 1979 doctoral dissertation [5]. The basic process of program slicing includes several key steps: first, construct the control flow graph (CFG) [6], which represents all possible execution paths in the program; then perform data flow analysis to determine how the values of variables and expressions propagate within the program; then, based on the analysis results, construct a dependency graph [7] and use slicing algorithms, such as forward slicing [8], backward slicing [9], static slicing [10], dynamic slicing [11], etc., to extract code segments related to requirements from the dependency graph.

B. Challenges

As an emerging system programming language, Rust’s unique ownership [2] and lifetime [3] checking mechanisms directly affect the slicability of code. In the Rust programming language, ownership and lifetimes are two core concepts for memory safety at compile time. Ownership rules define how resources are allocated, shared, and released, while lifetimes ensure the validity of references. When slicing, it is necessary to clarify the ownership of the data to avoid multiple releases of data or dangling references. At the same time, the lifetime of the slice must be consistent with the shortest lifetime of the referenced data. These factors combined bring new challenges to slicing Rust programs, leading to a relative scarcity of Rust slicing tools at present.

C. Current Status of Tools

In the Rust language, the abstract syntax tree (AST) plays an extremely important role. The AST represents various elements in the source code and their interrelationships in a tree-like form. By recursion or iteration, each node of the AST can be easily accessed, making it simple to traverse the

code. In Rust, the language’s AST can be obtained through the `syn` library [12]. The `Span` in the `syn` library is a core concept, representing a specific area in the source code, such as the start and end positions of an identifier, expression, or statement. This concept is crucial for understanding the code structure and performing precise code analysis.

Furthermore, `Kani` [13] is a symbolic model checking tool for the intermediate representation (MIR) [14] of Rust. It serves as a backend for the Rust compiler’s code generation, using the `kani::proof` attribute to specify functions that need to be verified with the `Kani` tool. `Kani` performs control flow and data flow analysis on these functions and transforms MIR into `goto` programs. `Goto` programs are a type of control flow graph that includes key information such as variable and type information, function call relationships, execution paths, and conditional judgments. However, it is worth noting that `Kani` does not support slicing functions and therefore cannot perform slicing analysis.

In addition, `CBMC` [15] is a static verification tool designed to identify potential errors in programs. `CBMC` can optimize `goto` programs, including dead code elimination [16], function inlining [17], `Goto` slicing [18], etc. The core mechanism of `Goto` slicing is to deeply analyze the Control Flow Graph (CFG) [6], identify and replace unreachable basic blocks with skip statements, which are ultimately removed. This process specifically targets functions or methods that will never be touched during program execution, as well as branches in conditional statements that are never satisfied. Although `CBMC` is a powerful tool, it is mainly designed for C and C++ languages and lacks support for features of other languages, such as Rust’s ownership and lifetime.

III. METHODOLOGY

Fig. 1 depicts the technical framework of `RustPruner`. It consists of three principal processes: source code processing, AST analysis, and slicing implementation. The remaining sections will provide an in-depth presentation of the specifics for each process.

A. Source Code Processing

Our tool can conduct dependency analysis on both individual Rust files and entire Rust projects. Single files typically do not import Rust libraries other than the standard library and only contain Rust source code, suitable for simple scenarios. On the other hand, project files consist of multiple source files, configuration files, and dependencies, managed by Cargo, suitable for the development of complex applications or libraries. Therefore, during dependency acquisition, the two inputs are processed separately to improve the accuracy and efficiency of the analysis.

We receive a Rust file (or project) for verification and start from each statement in the target function **proof**, which is the entry function for the Kani verification tool and also the slicing entry for RustPruner. Conduct a data flow analysis, using the Kani verification tool to obtain a control flow graph in the form of a goto file, which includes all possible execution paths and a large amount of dependency information unrelated to the proof framework. Then, conduct a backward analysis through the assertions in the proof or the output or assertions in the functions it calls. Using CBMC, mark the key information and remove the irrelevant information from these goto files to obtain a new goto file that only contains dependency information related to the proof framework. Finally, we extract the source code information corresponding to the key information from the new goto file, including the proof of the target function and all its related dependencies, forming a dependency relationship. A dependency relationship refers to a Rust element (such as a function, static variable, enumeration, etc.) that requires another Rust element to function properly. This relationship indicates that if the depended-upon element changes, the elements that depend on it may also need to change accordingly.

The following Rust code snippet is an example of demonstrating the function of each method.

```

1 struct Point {
2     x: i32,
3     y: i32,
4 }
5 impl Point {
6     fn distance(&self, other: &Point) -> i32
7     {
8         let dx = self.x - other.x;
9         let dy = self.y - other.y;
10        (dx.abs() + dy.abs()).max(0)
11    }
12 fn midpoint(p1: &Point, p2: &Point) -> Point
13 {
14     Point{x: (p1.x + p2.x) / 2, y: (p1.y + p2.y) / 2}
15 }
16 fn sum_of_coordinates(point: &Point) -> i32
17 {
18     point.x + point.y
19 }
20 fn main() {
21     let p1 = Point{x: 1, y: 2};
22     let p2 = Point{x: 5, y: 2};

```

```

21     let p3 = Point{x: 0, y: 0};
22     let dist = p1.distance(&p2);
23     assert!(dist == 4);
24 }
25 #[kani::proof]
26 fn verifymain() {
27     main();
28 }

```

The above source code can be parsed into the structure diagram shown in Fig. 2. As the Figure shows, each red arrow in the parsing diagram is obtained through source code processing. Starting from the entry flag `proof`, it enters the unique called function `main` of the function `verifymain()`, from `assert` it finds the definition statement of the variable `dist`, and then according to the definition statement, it finds the called function `distance` and parameters `p1` and `p2` in the right value of the statement, and finds the structure definition of the parameter `struct`. Finally, the nodes involved are connected with red arrows. The arrows in the diagram represent the dependency and dependent relationships, with the starting point of the arrow depending on the endpoint of the arrow.

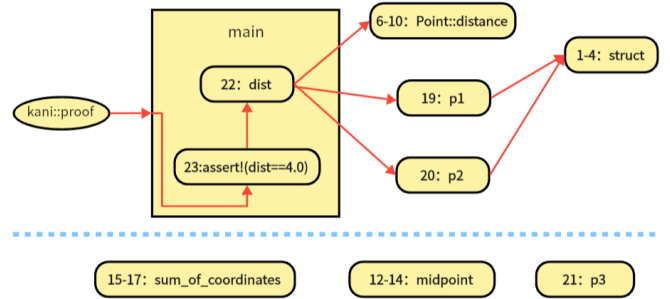


Fig. 2: Program Analysis

B. AST Analysis

Upon receiving a file (or project) for verification, we also conduct code parsing on the source files, which is done concurrently with dependency analysis. We compile the source code and construct an abstract syntax tree (AST) at the same time, traversing the AST to obtain the necessary structures and data from the source code.

Specifically, we have defined an information collector named `CodeCollector`, which traverses all files in the specified directory and filters out all Rust source files. For each Rust source file, it reads the file content and uses the `syn` library to parse the content into an abstract syntax tree (AST). Additionally, `CodeCollector` implements the visit trait, enabling it to customize the access and extraction of specific information within the code. To be precise, it obtains basic information of different types of Rust language elements from the AST, which is the Node information. Node information includes the name, type, starting line number, and ending line number, etc. The different types of Rust language elements include functions, static variables, enumerations, traits, and `impl` blocks, etc. All collected Node information is stored in the `CodeCollector`'s

information collector, forming a Rust element information repository, namely the Span library. This plays a crucial role in the subsequent code slicing process.

When dealing with Rust’s lifetime and ownership issues, the Node information is utilized. Each Node information covers the entire scope of Rust elements, essentially ensuring the complete architecture of the dependent elements. In this way, semantic boundaries, operational logic, and resource management are encapsulated within a closed context, thereby ensuring the correctness and reliability of the program. Thus, the internal code logic is complete, including the entire process of variable creation, usage, and destruction. At the same time, the paths of ownership transfer are not mistakenly truncated. So it is unlikely to accidentally remove key variables and create or destroy code when extracting code snippets, which can greatly reduce the risk of code violating ownership and lifetime rules after slicing.

As shown in Fig. 2, each yellow node is obtained through the AST analysis, representing different types of items in the Rust source code, including constants (Const), functions (Fn), structures (Struct), implementations (Impl), and assertions (assert) types, totaling 10 nodes. Each node contains corresponding information, including line number information, code structure type flags, source code paths, and other information.

C. Slicing Implementation

In the slicing implementation phase, we first traverse and filter the Span library within the organizational structure information. According to specific sorting rules, we extract the Nodes that match the dependency relationships from the library and map the source code portions of these Nodes to a new file. Specifically, for each function Node, we determine whether its starting and ending line numbers include the function line numbers collected in the dependency relationships. If the condition is met, we selectively write the function definitions into the new file. Similarly, we achieve the extraction and preservation of code snippets such as use statements, structures, and so on.

In Fig. 2, the blue dashed line is the function implemented in this part. The ones that match successfully will appear above the blue dividing line, and then be sorted and reorganized; the nodes that do not match successfully, that is, those irrelevant codes, will be below the blue dividing line, ready to be eliminated.

Therefore, by inputting the example source code into Rust-Pruner and proceeding through the aforementioned three steps, the following sliced code is obtained.

```

1 struct Point {
2     x: i32,
3     y: i32,
4 }
5 impl Point {
6     fn distance(&self, other: &Point) -> i32
7     {
8         let dx = self.x - other.x;
9         let dy = self.y - other.y;

```

```

9         (dx.abs() + dy.abs()).max(0)
10    }
11 }
12 fn main() {
13     let p1 = Point{x: 1, y: 2};
14     let p2 = Point{x: 5, y: 2};
15     let dist = p1.distance(&p2);
16     assert!(dist == 4);
17 }
18 #[kani::proof]
19 fn verify_main() {
20 }

```

IV. EXPERIMENTAL EVALUATION

Multiple experiments were conducted to evaluate the efficacy of program slicing for Rust applications. This section describes the experimental setup and experimental result.

A. Experimental Setup

The experimental setup includes the experimental environment, the test set used, the slicing criteria and an explanation of the benchmarks for comparison. All experiments were done on a machine equipped with AMD Ryzen 8845H CPUs, operating at 3.80 GHz, and configured with 32 GB of RAM. Each experiment was constrained to utilize only a single core and 8 GB of memory. The software environment for the experiments is described in Table 1.

TABLE I: Experimental Software Environment

Software	Version
Ubuntu	22.04.4 LTS
Rust	1.70.0-nightly
Kani	0.51.0
CBMC	cbmc-5.95.1 (64-bit x86_64 linux)

The test set for the experiments comprises four verification projects extracted from actual operating system projects. All are from the task scheduling module: Project A selects the first ready thread from the thread queue to be the current thread. Project B executes the current thread and switches between user mode and kernel mode. Project C performs cleanup operations based on the status of the thread that has finished running. Project D is the main loop of the user-mode program. Each project has an average of 1,079 source lines of code (sloc). The largest project contains 1,543 sloc, while the smallest includes 222 sloc computed by VS Code Counter¹.

Our slicing criterion involves all statements under the #[kani::proof] attribute. This attribute marks functions as entry points for Kani to verification, indicating to the Kani that the specified function contains assertions for formal verification. When Kani is executed, it focuses specifically on these marked functions, attempting to formally verify that the assertions hold true under all possible inputs. Our slicing tool retains all statements dependent on these functions. Through algorithmic processing tailored for Rust programs, it ensures that the sliced

¹<https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter>

program remains executable and that the verification results are consistent with the original program.

Prior to verifying Rust projects, we conducted a survey of many static slicing tools. Within our current research scope, no practical tools were found capable of slicing Rust programs directly. Consequently, we compiled Rust into LLVM code and tested a variety of tools that perform slicing on LLVM-IR or LLVM-bitcode. These tools included LLVM-slicing [19], LLVM-slicer², semslice [20], DG [21], and sbt-slicer³. However, we discovered that existing LLVM-level slicing tools do not support slicing of Rust programs effectively. Therefore, our experiments primarily focus on evaluating the performance of our tool rather than comparing it with other existing tools.

B. Experimental Result

We conducted experiments on the aforementioned test set by measuring the slicing time, verification time and the size of the slice. All the algorithms and ideas described in this paper have been implemented to ensure the feasibility of slicing Rust programs and to maintain the executability and verifiability of the programs after slicing.

TABLE II: Verification Results

Project	Verification Consistency	Failures	Success
A	True	1	0
B	True	0	3
C	True	0	1
D	True	0	1

After slicing, the projects maintain consistency with the original projects in terms of verification attributes, as shown in TABLE II. Project A has 1 failure, Project B has 3 successfully verified harnesses, and both Project C and Project D have 1 successfully verified harness each. The summary of the experimental results is shown in TABLE III. The descriptions for some columns in TABLE III are as follows:

- **Ver. Time:** Average verification time of the original project
- **Src Size:** Number of source lines of code in the original project
- **Slice Size:** Number of source lines of code after project slicing
- **Slice Time:** Average slicing time for the project

The methodology adopted for measuring performance included the following procedure: Each time measurement, whether for project verification or a specific slice, was performed 10 times. The initial iteration was always discarded to eliminate the effects of dynamically loading libraries into physical memory and data remaining in the disk cache. After each slicing or verification, the project’s object files were cleared before conducting the experiment again.

TABLE III and Fig. 3 provide a comprehensive analysis of the slicing effects on various projects labeled A, B, C, and D.

²<https://github.com/jirislaby/LLVMSlicer>

³<https://github.com/staticafi/sbt-slicer>

TABLE III: experimental statistics

Proj.	Ver. Time	Src Size	Slice Size	Slice Time
A	0.98 s	222 sloc	186 sloc	1.25 ms
B	2.17 s	1013 sloc	761 sloc	7.89 ms
C	2.97 s	1543 sloc	978 sloc	11.26 ms
D	3.67 s	1538 sloc	1316 sloc	24.96 ms

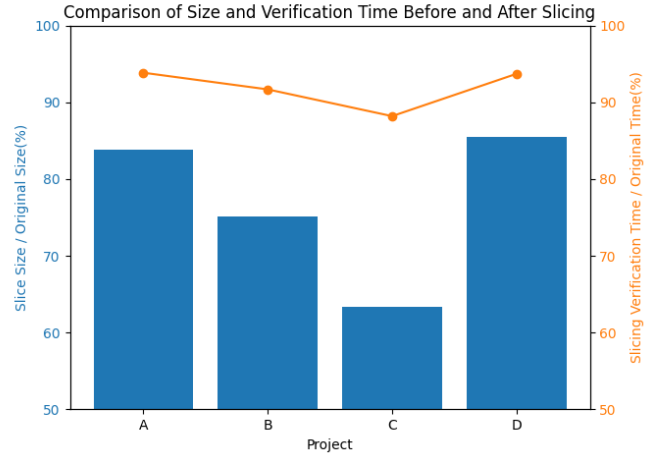


Fig. 3: Comparison of Size and Verification Time Before and After Slicing

Each project underwent a series of measurements that assessed slicing efficiency based on source size reduction before and after slicing.

During the course of the experiment, Proj. A demonstrated the smallest reduction in source code size but was processed the fastest, indicating efficiency in handling smaller-scale projects. In contrast, Proj. B, with a larger original source size, showed significant reductions in source lines but required longer processing times, reflecting the increased complexity when dealing with large codebases.

Particularly, Proj. C and Proj. D, which started with larger source sizes, exhibited substantial reductions post-slicing, validating the effectiveness of the slicing tool in handling larger projects. At the same time, the reduction in size and the decrease in verification time after slicing may indicate a positive correlation: that is, the reduction in project size helps to speed up the verification process.

These findings demonstrate that the RustPruner tool has successfully extended the application of program slicing to Rust projects, ensuring, to some extent, that the slicing outcomes are executable and that the verification results are consistent.

V. RELATED WORK

Program slicing can be categorized based on the consideration of specific inputs to the program into static slicing [5] and dynamic slicing [22]. Static slicing [5] involves analysis directly from the program’s source code, employing techniques such as data flow and control flow analysis to extract segments of code related to a particular point of interest. This approach

takes into account all possible executions of the program, yielding a more comprehensive slicing result. In contrast, dynamic slicing [22] is conducted during the actual execution of the program, where relevant code segments are extracted based on the program's execution path and state information. Dynamic slicing is more readily implementable and tends to produce more precise slicing outcomes.

Program slicing is distinguished by the direction of analysis relative to a variable of interest, categorized into forward slicing [8] and backward slicing [9]. Forward slicing [8] initiates at a designated starting point, such as a specific variable or statement, and proceeds in the direction of the program's control and data flow to identify all code segments that could potentially influence this point of origin. Conversely, backward slicing [9] commences at a specified endpoint, like a particular output or error, and retraces the control and data flow pathways to extract code segments that may have an impact on this terminal point.

Program slicing can be differentiated by the analytical techniques and methodologies employed, which include Weiser's original slicing method [5], methods based on program information flow relationships [23], and graph reachability-based slicing methods [24]. Weiser's original slicing method [5] centers on scrutinizing the trajectory of data within a program and ascertaining the statements that exert influence over the data's values. Methods based on program information flow relationships [23] concentrate on the interdependencies among program statements, employing a syntax-directed, bottom-up methodology to compute these dependencies. Conversely, graph reachability-based slicing methods [24] spotlight the dependencies and reachability among program statements. By fabricating distinct dependency graphs, which encapsulate a variety of relational contexts, a spectrum of slicing outcomes can be realized.

Program slicing methodologies are classified into the Program Dependence Graph (PDG) [25] and the System Dependence Graph (SDG) [26] based on their scope of analysis. PDG [25] is a graphical representation of a program that captures control and data dependencies, serving as a foundation for intraprocedural slicing by enabling the precise extraction of code segments related to specific program points. Building upon this, the SDG [26] extends the scope to interprocedural slicing by incorporating dependencies across multiple procedures, thus accounting for the broader context of a system's behavior.

Our methodology adeptly applies the graph reachability approach through the utilization of PDG, conducting a rigorous intraprocedural analysis. We introduce a novel custom visitor for traversing AST, which efficiently identifies and collects critical code elements. By focusing on backward, static slicing and capitalizing on the strengths of graph reachability, our research effectively surpasses the limitations of current methods. RustPruner's providing a refined and effective slicing technique that bolsters the analytical capabilities of software engineering practitioners and researchers alike.

REFERENCES

- [1] Silva and Josep, "A vocabulary of program slicing-based techniques," *ACM computing surveys*, pp. 12(1–41), 2012.
- [2] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proceedings of the ACM on Programming Languages*, pp. 1224–1252, 2023.
- [3] D. Blaser, "Simple explanation of complex lifetime errors in rust," *ETH Zürich*, 2019.
- [4] Y. Kashima, T. Ishio, and K. Inoue, "Comparison of backward slicing techniques for java," *IEICE Trans. Inf. Syst.*, vol. 98, no. 1, pp. 119–130, 2015.
- [5] W. M., "Program slices : Formal, psychological, and practical investigations of an automatic program abstraction method," *PhD thesis, University of Michigan*, 1979.
- [6] F. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, pp. 1–19, 1970.
- [7] S. Liang and L. Larson, "Slicing objects using system dependence graphs," pp. 358–367, November 1998.
- [8] K. Awad, M. Abdallah, A. Tamimi, A. Ngah, and H. Tamimi, "A proposed forward clause slicing application," *Indonesian Journal of Electrical Engineering and Computer Science*, pp. 1–6, 2019.
- [9] B. Alokush, M. Abdallah, and M. Alrifae, "A proposed java static slicing approach," *Indonesian Journal of Electrical Engineering and Computer Science*, pp. 308–317, 2018.
- [10] X. Zhaogui, Q. Ju, C. Lin, C. Zhifei, and X. Baowen, "Static slicing for python first-class objects," *13th International Conference on Quality Software*, pp. 117–124, 2013.
- [11] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, pp. 187–195, 1990.
- [12] The crate syn. <https://crates.io/crates/syn>. Accessed: 2024-05-11.
- [13] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in rust," in *ICSE-SEIP '22: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022.
- [14] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *CCS '21: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [15] D. Kröning and M. Tautschnig, "CBMC: The c bounded model checker," *Proc. TACAS*, pp. 389–391, 2014.
- [16] T. T., R. M., and S. Z., "Finding missed optimizations through the lens of dead code elimination," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 697–709, 2022.
- [17] P. Zhao and J. Amaral, "Function outlining and partial inlining," in *17th International Symposium on Computer Architecture and High Performance Computing*, 2005.
- [18] J.-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1097–1113, 1994.
- [19] Y.-Z. Zhang, "Sympas: Symbolic program slicing," *J. Comput. Sci. Technol.*, p. 397–418, apr 2021.
- [20] B. Beckert, T. Bormer, S. Gocht, M. Herda, D. Lentzsch, and M. Ulbrich, "Semslice: Exploiting relational verification for automatic program slicing," in *Integrated Formal Methods. 13th International Conference, IFM*, 2017.
- [21] M. Chalupa, "Dg: Analysis and slicing of llvm bytecode," in *Automated Technology for Verification and Analysis*, ser. LNCS, vol. 12302. Springer, 2020.
- [22] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, pp. 187–195, 1990.
- [23] J.-F. Bergeretti and B. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 37–61, 1985.
- [24] X. Qi and B. Xu, "An approach to slicing concurrent ada programs based on program reachability graphs," *International Journal of Computer Science and Network Security*, pp. 29–37, 2006.
- [25] K. Ottenstein and L. Ottenstein, "The program dependence graph in a software development environment," *ACM SIGPLAN Notices*, pp. 177–184, 1984.
- [26] S. Horwitz and T. Reps, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 26–60, 1990.