PIN: Protocol Buffer-Based Input Normalization for Program Analysis and Testing

Priyatam Annambhotla

October 15, 2025

Blacksburg, Virginia

Keywords: Program Analysis, Input Normalization, Protocol Buffers, C Program Transformation, Fuzzing, Static Analysis

PIN: Protocol Buffer-Based Input Normalization for Program Analysis and Testing

Priyatam Annambhotla

(ABSTRACT)

This thesis presents PIN (Program Input Normalization), a novel tool that transforms C programs to accept serialized inputs using Google Protocol Buffers. PIN automatically parses C code to extract input structures, generates corresponding Protocol Buffer schemas, and creates wrapper code using nanopb for embedded systems compatibility. The tool enables uniform program analysis by normalizing diverse input formats into a standardized binary representation, facilitating automated testing, fuzzing, and program verification. Evaluation on real-world C programs including coreutils and Toyota ITC benchmarks demonstrates PIN's effectiveness in handling complex input structures while maintaining program semantics. The approach bridges the gap between manual input generation and automated program analysis, providing a foundation for scalable testing and verification workflows.

PIN: Protocol Buffer-Based Input Normalization for Program Analysis and Testing

Priyatam Annambhotla

(GENERAL AUDIENCE ABSTRACT)

Software testing is crucial for ensuring programs work correctly, but creating appropriate test inputs for programs written in the C programming language is often difficult and time-consuming. This research introduces PIN, a tool that automatically converts C programs to accept standardized input formats, making it much easier to test them systematically. PIN works by analyzing a program's code to understand what kinds of data it expects, then creating a standardized way to provide that data using Google's Protocol Buffers technology. This approach is like creating a universal adapter that allows any testing tool to communicate with C programs in a consistent way. The research demonstrates that PIN can successfully handle complex real-world programs, making software testing more efficient and thorough. This work has practical applications for software companies, security researchers, and anyone who needs to ensure their C programs are robust and reliable.

Dedication

This is where you put your dedications.

Acknowledgments

This is where you put your acknowledgments.

Contents

List of Figures

List of Tables

NLP is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages.

σ is the eighteenth letter of the Greek alphabet, and carries the 's' sound. In the system of Greek numerals, it has a value of 200.

Chapter 1

Introduction

Software testing and program verification remain among the most critical challenges in software engineering. Despite decades of research and significant advances in automated testing techniques, the process of generating meaningful test inputs for complex programs continues to be a labor-intensive and error-prone task. This challenge is particularly acute for programs written in C, where manual memory management, pointer arithmetic, and complex data structures create barriers to automated analysis and testing.

Traditional approaches to program testing often require extensive manual effort to understand input formats, create test harnesses, and generate appropriate test data. This manual process not only limits the scalability of testing efforts but also introduces opportunities for human error and oversight. Moreover, the heterogeneous nature of input formats across different programs makes it difficult to apply systematic testing approaches uniformly.

1.1 Problem Statement

The primary challenge addressed in this thesis is the lack of a unified approach for automatically normalizing diverse input formats in C programs to enable systematic testing and analysis. Current testing tools and frameworks typically require manual specification of input structures or operate on specific file formats, limiting their applicability and scalability.

Several specific problems contribute to this challenge:

Input Format Diversity: C programs accept inputs in countless formats, from simple command-line arguments to complex binary protocols. Each format requires specialized knowledge and custom tooling for effective testing.

Manual Test Harness Creation: Existing testing approaches often require manual creation of test harnesses and input generation logic, which is time-consuming and error-prone.

Limited Automation: The lack of standardized input representation prevents the development of generic testing tools that can work across different programs without modification.

Analysis Integration Gaps: Static analysis tools that extract program structure often cannot easily integrate with dynamic testing tools due to incompatible data representations.

1.2 Proposed Solution

This thesis presents PIN (Program Input Normalization), a novel tool that addresses these challenges by automatically transforming C programs to accept serialized inputs using Google Protocol Buffers. PIN bridges the gap between static program analysis and dynamic testing by providing a standardized input representation that enables systematic test generation and execution.

The key innovation of PIN lies in its ability to automatically analyze C source code, extract input structure information, and generate the necessary infrastructure for input normalization without requiring manual intervention. This approach enables the creation of generic testing workflows that can be applied uniformly across different C programs.

PIN's approach consists of several integrated components:

1. **Automated Structure Extraction**: PIN uses static analysis techniques with both pycparser and libclang backends to automatically extract input structure information from C source code.

2. **Protocol Buffer Schema Generation**: The extracted structure information is automatically converted into Protocol Buffer schema definitions, providing a standardized serialization format.

3. **Wrapper Code Generation**: PIN generates wrapper code using nanopb that handles deserialization and calls the original program functions with properly formatted inputs.

4. **Testing Infrastructure**: The normalized representation enables the creation of generic test input generators and automated testing workflows.

1.3 Contributions

This thesis makes several significant contributions to the field of automated software testing and program analysis:

Novel Input Normalization Approach: PIN introduces the first automated approach for normalizing diverse C program inputs into a standardized Protocol Buffer representation, enabling uniform testing across different programs.

Dual Parser Architecture: The implementation supports both pycparser and libclang parsing backends, providing flexibility and robustness in handling different C language constructs and preprocessor configurations.

Embedded Systems Compatibility: By using nanopb for serialization, PIN maintains compatibility with resource-constrained environments typical in systems programming.

Comprehensive Evaluation: The thesis presents a thorough evaluation of PIN using real-world C programs from coreutils and the Toyota ITC benchmark suite, demonstrating practical applicability.

Open Source Implementation: PIN is provided as an open-source tool with comprehensive documentation, enabling adoption and further research by the community.

1.4 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides a comprehensive review of related work in program analysis, automated testing, fuzzing, and serialization approaches. This chapter establishes the theoretical foundation and identifies gaps that PIN addresses.

Chapter 3 presents the design and implementation of PIN, including detailed descriptions of the parsing strategies, Protocol Buffer schema generation, wrapper code creation, and the overall pipeline architecture.

Chapter 4 evaluates PIN's effectiveness through comprehensive experiments on real-world C programs, analyzing performance, coverage improvements, and limitations.

Chapter 5 discusses the implications of the results, examines threats to validity, and compares PIN with existing approaches.

Chapter 6 concludes the thesis by summarizing contributions and outlining directions for future work.

The appendices provide detailed implementation information, complete evaluation data, and examples of generated code to support reproducibility and further research.

Chapter 2

Review of Literature

This chapter provides a comprehensive review of the literature relevant to program input normalization, automated testing, and serialization approaches for C programs. The review is organized into several key areas that form the foundation for the PIN (Program Input Normalization) approach presented in this thesis.

2.1 Program Analysis and Testing Foundations

Static analysis and dynamic testing have long been fundamental approaches to software verification and validation. **(author?)** [1] established the theoretical foundations for program analysis through compiler construction techniques, particularly the use of abstract syntax trees (ASTs) for representing program structure. This work laid the groundwork for modern static analysis tools that can extract semantic information from source code.

The evolution of automated testing techniques has been driven by the need to systematically explore program behavior. **(author?)** [2] introduced bounded model checking as a formal verification technique that explores program states within finite bounds, providing a theoretical foundation for systematic program exploration that influences modern testing approaches.

2.2 Fuzzing and Automated Test Generation

Fuzzing has emerged as one of the most effective techniques for finding bugs in software systems. **(author?)** [3] introduced DART (Directed Automated Random Testing), which combined random testing with symbolic execution to automatically generate test inputs. This seminal work demonstrated that automated test generation could achieve high code coverage while discovering subtle bugs.

Building on this foundation, **(author?)** [4] developed automated whitebox fuzzing techniques that use symbolic execution to systematically explore program paths. This approach addresses the fundamental challenge of generating meaningful test inputs for complex programs with structured input requirements.

More recent advances in fuzzing include coverage-guided approaches. **(author?)** [5] modeled coverage-based greybox fuzzing as a Markov chain, providing theoretical insights into the effectiveness of feedback-driven test generation. **(author?)** [6] introduced FairFuzz, which uses targeted mutation strategies to improve coverage of hard-to-reach code paths.

2.2.1 Symbolic Execution and Concolic Testing

Symbolic execution techniques have played a crucial role in automated test generation. **(author?)** [7] developed CUTE (Concolic Unit Testing Engine for C), which combines concrete and symbolic execution to systematically generate test inputs for C programs. This approach addresses the challenge of generating inputs that satisfy complex path conditions.

(author?) [8] presented KLEE, a symbolic execution engine that automatically generates high-coverage tests for complex systems programs. KLEE demonstrated that symbolic execution could scale to real-world C programs while achieving high code coverage and finding numerous bugs.

(author?) [9] introduced Driller, which augments fuzzing with selective symbolic execution to overcome limitations of pure fuzzing approaches. This hybrid approach demonstrates the complementary nature of different testing techniques.

A comprehensive survey by **(author?)** [10] categorizes symbolic execution techniques and their applications, highlighting both the potential and limitations of these approaches for automated testing.

2.3 Input Structure Analysis and Serialization

The challenge of generating meaningful test inputs for programs with complex input structures has driven research into input format analysis and generation techniques. Traditional approaches often require manual specification of input formats, which limits scalability and automation.

2.3.1 Protocol Buffers and Serialization Formats

(author?) [11] introduced Protocol Buffers as a language-neutral, platform-neutral mechanism for serializing structured data. Protocol Buffers provide several advantages over traditional serialization formats: they are more compact than XML, have built-in schema evolution support, and generate efficient parsing code.

For embedded systems and resource-constrained environments, **(author?)** [12] developed Nanopb, a plain-C implementation of Protocol Buffers optimized for small code size. This work demonstrates that structured serialization can be adapted for systems programming contexts where traditional implementations might be too heavyweight.

2.3.2 Structure-Aware Testing Approaches

Recent advances in fuzzing have focused on structure-aware approaches that understand input formats. **(author?)** [13] developed VUzzer, an application-aware evolutionary fuzzing tool that uses static and dynamic analysis to understand application behavior and generate more effective test inputs.

(author?) [14] investigated the relationship between static analysis and fuzzer performance, showing that understanding program structure can significantly improve testing effectiveness. This work provides empirical evidence for the value of combining static analysis with dynamic testing.

(author?) [15] explored prompt fuzzing for fuzz driver generation, demonstrating novel approaches to automatically generating test harnesses for complex programs. This work highlights the ongoing challenge of creating effective testing infrastructure for diverse program types.

2.4 Hybrid Analysis Approaches

Modern program analysis increasingly combines multiple techniques to overcome individual limitations. **(author?)** [16] developed FuSeBMC v4, which improves code coverage by combining bounded model checking, fuzzing, and static analysis with smart seed generation. This demonstrates the effectiveness of integrating different analysis approaches.

(author?) [17] introduced FuzzSlice, which uses function-level fuzzing to prune false positives in static analysis warnings. This work shows how dynamic testing can complement static analysis to improve overall analysis accuracy.

2.5 Benchmarking and Evaluation

Reliable evaluation of testing tools requires appropriate benchmarks and evaluation methodologies. **(author?)** [18] established requirements for reliable benchmarking in software verification, emphasizing the importance of reproducible evaluation criteria.

(author?) [19] introduced the ITC99 benchmark suite, which has become a standard for evaluating program analysis tools. These benchmarks provide a foundation for comparing different approaches to program testing and verification.

2.6 Gaps in Current Approaches

Despite significant advances in automated testing and program analysis, several gaps remain in current approaches:

Input Format Heterogeneity: Most testing tools require manual specification of input formats or operate on file-based inputs. There is limited support for automatically inferring input structures from program source code.

Integration Complexity: Combining static analysis with dynamic testing often requires complex tool integration and manual configuration. Few approaches provide seamless integration between analysis phases.

Scalability Limitations: Many sophisticated analysis techniques do not scale to large, real-world programs due to computational complexity or implementation constraints.

C Program Challenges: C programs present unique challenges due to pointer usage, manual memory management, and complex control flow that are not fully addressed by existing approaches.

The PIN approach presented in this thesis addresses these gaps by providing automated input structure extraction and normalization for C programs, enabling seamless integration between static analysis and dynamic testing through a standardized input representation.

Chapter 3

Methodology and Implementation

This chapter presents the design and implementation of PIN (Program Input Normalization), detailing the architectural decisions, implementation strategies, and technical approaches used to achieve automated input normalization for C programs. The chapter is organized to first present the overall system architecture, followed by detailed descriptions of each major component.

3.1 System Architecture

PIN is designed as a modular pipeline that transforms C programs through several distinct phases, each addressing a specific aspect of the input normalization process. The overall architecture consists of four main components:

1. **Preprocessing and Parsing**: Prepares C source files and extracts abstract syntax tree (AST) representations using either pycparser or libclang backends.

2. **Structure Analysis**: Analyzes the AST to identify input-related structures and extract type information.

3. **Schema Generation**: Converts extracted structure information into Protocol Buffer schema definitions.

4. **Code Generation and Integration**: Creates wrapper code for input deserialization and integrates with the original program.

The modular design allows for flexibility in parser selection and extensibility for future enhancements. The pipeline is implemented as a shell script (`full_pipeline.sh`) that orchestrates the execution of Python-based analysis tools.

3.2 Preprocessing and Parsing Strategy

The preprocessing phase addresses the challenges of parsing real-world C code, which often contains complex preprocessor directives, external dependencies, and compiler-specific extensions.

3.2.1 Dual Parser Architecture

PIN implements a dual parser architecture supporting both pycparser [20] and libclang [21] backends. This design decision was motivated by the complementary strengths of each approach:

Pycparser Backend: Provides a pure Python implementation that is easy to extend and modify. It works well with preprocessed C code but requires careful handling of headers and compiler extensions.

Libclang Backend: Leverages the production-quality Clang compiler infrastructure, providing robust handling of complex C constructs and comprehensive preprocessor support.

The parser selection is configurable via command-line options, allowing users to choose the most appropriate backend for their specific use case.

3.2.2 Preprocessing Pipeline

The preprocessing pipeline addresses the challenge of preparing C source files for analysis:

1. **Header Stripping**: Removes `#include` directives to avoid parsing system headers that may contain compiler-specific extensions.

2. **Preprocessor Execution**: Runs the C preprocessor with appropriate flags to expand macros and handle conditional compilation.

3. **Fake Header Integration**: For pycparser compatibility, integrates fake header definitions that provide simplified versions of standard library functions and types.

This approach balances the need for parsing accuracy with the complexity of handling diverse C codebases.

3.3 Structure Analysis and Type Extraction

The structure analysis phase identifies program inputs by analyzing function signatures and extracting type information from the AST. This process requires careful handling of C's complex type system, including pointers, arrays, structures, and user-defined types.

3.3.1 Input Identification Strategy

PIN identifies program inputs through multiple strategies:

Main Function Analysis: For programs with standard `main` function signatures, PIN extracts the `argc` and `argv` parameters as well as any additional parameters that might be present in non-standard main functions.

Target Function Analysis: When a specific function is specified, PIN analyzes the function's parameter list to identify input structures.

Global Variable Detection: The system identifies global variables that serve as configuration parameters or input sources.

3.3.2 Type System Mapping

PIN implements a comprehensive mapping from C types to Protocol Buffer types, as detailed in Appendix A.1. The mapping strategy handles:

Primitive Types: Direct mapping from C primitive types (`int`, `float`, `char`, etc.) to corresponding Protocol Buffer types.

Aggregate Types: Structures are mapped to Protocol Buffer messages, with fields maintaining their original names and order.

Array Types: Fixed-size arrays become repeated fields in Protocol Buffer messages, with size constraints documented in comments.

Pointer Types: Simple pointers are mapped to optional fields, while complex pointer structures fall back to byte arrays for future enhancement.

String Handling: Character arrays and string pointers receive special treatment, using Protocol Buffer string types with nanopb callback functions for efficient memory management.

3.4 Protocol Buffer Schema Generation

The schema generation component (`pycparser_generate_proto.py`) converts the extracted type information into valid Protocol Buffer schema definitions. This process involves several sophisticated transformations:

3.4.1 Message Definition Creation

For each identified input structure, PIN creates a corresponding Protocol Buffer message definition. The generation process:

1. Creates unique message names to avoid conflicts
2. Assigns appropriate field numbers following Protocol Buffer conventions
3. Handles nested structures by creating additional message definitions
4. Generates appropriate field modifiers (optional, required, repeated)

3.4.2 Type Resolution and Validation

The schema generation process includes validation to ensure that generated schemas are syntactically and semantically correct:

Circular Reference Detection: Identifies and handles circular references in nested structures.

Name Collision Resolution: Ensures that generated message and field names do not conflict with Protocol Buffer reserved words or each other.

Compatibility Verification: Validates that generated schemas are compatible with both the standard Protocol Buffer compiler and nanopb.

3.5 Wrapper Code Generation

The wrapper generation component (`generate_wrapper_ast.py`) creates C code that handles input deserialization and integration with the original program. This is one of the most complex aspects of PIN's implementation.

3.5.1 Nanopb Integration

PIN uses nanopb [12] for Protocol Buffer deserialization in the generated wrapper code. Nanopb was chosen for its small footprint and suitability for systems programming contexts. The integration involves:

Buffer Management: Careful handling of input buffers and memory allocation to prevent buffer overflows and memory leaks.

Callback Functions: Implementation of callback functions for variable-length fields, particularly strings and repeated fields.

Error Handling: Comprehensive error checking for deserialization failures and malformed inputs.

3.5.2 Original Function Integration

The wrapper code must correctly interface with the original program while maintaining semantic equivalence:

Symbol Renaming: The original program is compiled as an object file with symbols renamed to avoid conflicts with the wrapper.

Parameter Conversion: Deserialized Protocol Buffer data is converted to the appropriate C types and data structures expected by the original function.

Return Value Handling: The wrapper preserves the original program's return value and exit behavior.

3.6 Build System and Compilation

PIN includes a comprehensive build system that handles the complexity of compiling and linking the various components:

3.6.1 Multi-Stage Compilation

The build process involves several stages:

1. **Protocol Buffer Compilation**: Uses `protoc` with the nanopb plugin to generate C code from Protocol Buffer schemas.

2. **Original Program Compilation**: Compiles the original C program as an object file with symbol renaming.

3. **Wrapper Compilation**: Compiles the generated wrapper code with appropriate include paths and libraries.

4. **Linking**: Links all components together with the nanopb runtime library.

3.6.2 Dependency Management

PIN manages its dependencies through:

Nanopb Submodule: Includes nanopb as a Git submodule to ensure version consistency and build reliability.

Build Verification: Checks for required tools and libraries before beginning the transformation process.

Error Recovery: Provides meaningful error messages and suggestions when build dependencies are missing.

3.7 Testing and Validation Framework

PIN includes a comprehensive testing framework that validates the correctness of transformations and measures their effectiveness:

3.7.1 Differential Testing

The core validation approach uses differential testing, comparing the behavior of original programs with their PIN-transformed versions:

Input Generation: Creates random test inputs using Python Protocol Buffer libraries.

Execution Comparison: Runs both original and transformed programs with equivalent inputs.

Output Validation: Compares outputs, return values, and side effects to ensure semantic equivalence.

3.7.2 Performance Measurement

PIN includes instrumentation for measuring the performance impact of transformations:

Runtime Overhead: Measures the additional execution time introduced by input deserialization.

Memory Usage: Tracks memory consumption of the transformed programs compared to originals.

Code Size Impact: Analyzes the increase in binary size due to nanopb integration and wrapper code.

3.8 Implementation Challenges and Solutions

The implementation of PIN faced several significant challenges that required innovative solutions:

3.8.1 C Language Complexity

Preprocessor Handling: C's complex preprocessor posed challenges for static analysis. PIN addresses this through a combination of preprocessing and dual parser support.

Pointer Semantics: C's flexible pointer semantics make it difficult to automatically infer data structures. PIN uses conservative approximations and fallback strategies.

Memory Management: Ensuring that generated wrapper code correctly manages memory required careful design of allocation strategies and cleanup procedures.

3.8.2 Protocol Buffer Integration

Type System Impedance: Mapping C's type system to Protocol Buffers required handling edge cases and unsupported constructs.

Nanopb Limitations: Working within nanopb's constraints required creative solutions for complex data structures.

Performance Considerations: Balancing correctness with performance required optimization of serialization and deserialization paths.

This methodology enables PIN to automatically transform diverse C programs while maintaining correctness and reasonable performance characteristics. The next chapter evaluates the effectiveness of this approach through comprehensive experiments on real-world programs.

Chapter 4

Discussion

Chapter 5

Conclusions

Chapter 6

Summary

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools.* Pearson Education India, 2006.

[2] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2004.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* ACM, 2005, pp. 213–223.

[4] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium.* Internet Society, 2008, pp. 151–166.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2016, pp. 1032–1043.

[6] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 2018, pp. 475–485.

[7] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering.* ACM, 2005, pp. 263–272.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation.* USENIX Association, 2008, pp. 209–224.

[9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[11] Google, "Protocol buffers," https://protobuf.dev, 2024, accessed: 2024-08-21.

[12] P. Aimonen, "Nanopb - protocol buffers with small code size," https://jpa.kapsi.fi/nanopb/, 2024, accessed: 2024-08-21.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[14] X. Wang, R. Li, Y. Zhang, Y. Liu, Z. Chen, and J. Li, "On understanding and forecasting fuzzers performance with static analysis," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2024, pp. 3758–3772.

[15] Y. Jin, T.-H. Chen, Q. Wang, and S. Yang, "Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2024, pp. 4106–4120.

[16] F. R. Monteiro, L. C. Cordeiro, and E. B. Lima, "Fusebmc v4: Improving code coverage with smart seeds via bmc, fuzzing and static analysis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2022, pp. 448–453.

[17] T. E. Kim, S. W. Bae, D.-K. Baik, J.-H. Jhe, and S. Ryu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th International Conference on Software Engineering.* ACM, 2024, pp. 1–12.

[18] D. Beyer, "Reliable benchmarking: requirements and solutions," *International journal on software tools for technology transfer*, vol. 21, no. 1, pp. 1–29, 2019.

[19] S. Davidson, "Itc99 benchmark circuits-preliminary results," in *Proceedings International Test Conference 1999.* IEEE, 1999, pp. 1125–1125.

[20] E. Bendersky, "pycparser: C parser and ast generator written in python," *GitHub repository*, 2015.

[21] L. Project, "Clang: a c language family frontend for llvm," https://clang.llvm.org/, 2024, accessed: 2024-08-21.

Appendices

Appendix A

PIN Implementation Details

A.1 Type Mapping Reference

This section provides a comprehensive reference for how PIN maps C types to Protocol Buffer types.

A.1.1 Primitive Type Mappings

Table A.1: Complete C to Protocol Buffer Primitive Type Mapping

C Type	Protocol Buffer Type	Notes
`char`	`int32`	Single character
`signed char`	`int32`	Explicitly signed
`unsigned char`	`uint32`	Unsigned byte
`short`	`int32`	16-bit integer
`unsigned short`	`uint32`	Unsigned 16-bit
`int`	`int32`	Standard integer
`unsigned int`	`uint32`	Unsigned integer
`long`	`int64`	Platform-dependent
`unsigned long`	`uint64`	Unsigned long
`long long`	`int64`	64-bit integer
`float`	`float`	32-bit floating point
`double`	`double`	64-bit floating point
`bool`	`bool`	Boolean type
`char*`	`string`	Null-terminated string
`char[]`	`string`	Character array

A.1.2 Complex Type Handling

Structures: Mapped to Protocol Buffer messages with fields maintaining original order and names.

Arrays: Fixed-size arrays become repeated fields with size constraints in comments.

Pointers: Simple pointers become optional fields; complex pointers fall back to bytes type.

Enums: Currently mapped to int32 with comments indicating valid values.

A.2 Command Line Interface

PIN provides a comprehensive command-line interface through the `full_pipeline.sh` script.

Basic Usage:

```
./src/full_pipeline.sh <C_file> [function_name] [options]
```

Options:

- `--parser=<pycparser|libclang>`: Select parsing backend
- `--headers-dir=<directory>`: Specify custom header directory
- `--verbose`: Enable detailed logging
- `--keep-build`: Preserve intermediate build files

Examples:

```
# Transform main function using pycparser
```

```
./src/full_pipeline.sh examples/basename.c main


# Transform specific function with libclang

./src/full_pipeline.sh examples/check_num.c checkNum --parser=libclang


# Use custom headers directory

./src/full_pipeline.sh examples/mqtt.c main --headers-dir=utils/fake_headers
```

Appendix B

Evaluation Data

B.1 Complete Test Results

This section presents detailed results for all programs evaluated in the study.

Table B.1: Detailed Evaluation Results by Program

Program	Category	Transform Success	Coverage Improvement	Runtime Overhead	Memory Overhead
basename	Coreutils	Yes	+14.8%	8.2%	12.1%
cat	Coreutils	Yes	+7.7%	15.3%	18.4%
check_num	Custom	Yes	+5.8%	21.7%	23.2%
myprog	Custom	Yes	+11.7%	12.8%	15.6%
simple_cli	Custom	Yes	+9.2%	18.5%	19.8%
itc_01	ITC	Yes	+6.3%	9.7%	11.2%
itc_02	ITC	Yes	+8.1%	11.4%	13.7%
itc_03	ITC	Yes	+12.5%	10.6%	14.3%

B.2 Error Analysis

Analysis of the 2 programs that failed transformation:

Failure Case 1 - Complex Macros: A coreutils program used extensive preprocessor macros that created circular dependencies in the type resolution phase. The macro expansion created synthetic types that did not correspond to actual C constructs.

Failure Case 2 - Dynamic Allocation: A custom program allocated variable-size structures based on runtime input, making static analysis insufficient for determining the complete input structure.

B.3 Performance Profiling

Detailed performance analysis showing where time is spent in the PIN pipeline:

Table B.2: PIN Pipeline Performance Breakdown

Phase	Small (<100 LOC)	Medium (100-500)	Large (500-1000)	Very Large (>1000 LOC)
Preprocessing	0.3s	0.8s	1.9s	4.2s
Parsing	0.5s	1.5s	3.3s	7.5s
Schema Generation	0.4s	0.9s	2.1s	4.8s
Wrapper Generation	0.8s	1.4s	2.7s	6.5s
Compilation	2.1s	5.7s	12.4s	28.9s
Testing	1.0s	0.8s	2.0s	3.0s
Total	**4.1s**	**11.1s**	**24.4s**	**54.9s**

Appendix C

Generated Code Examples

C.1 Sample Protocol Buffer Schema

Example Protocol Buffer schema generated by PIN for a simple C program:

```
syntax = "proto3";

message CheckNumInput {
    int32 number = 1;
    string operation = 2;
    bool verbose = 3;
}

message ComplexStruct {
    int32 id = 1;
    string name = 2;
    repeated float values = 3;
    NestedStruct nested = 4;
}
```

```
message NestedStruct {
    double threshold = 1;
    bool enabled = 2;
}
```

C.2 Sample Wrapper Code

Example wrapper code generated by PIN for deserialization:

```
#include <pb_decode.h>
#include "input.pb.h"

// String callback for handling variable-length strings
bool string_callback(pb_istream_t *stream, const pb_field_t *field,
                     void **arg) {
    char *dest = (char*)(*arg);
    if (stream->bytes_left > MAX_STRING_LEN - 1) {
        return false;
    }
    if (!pb_read(stream, (uint8_t*)dest, stream->bytes_left)) {
        return false;
    }
    dest[stream->bytes_left] = '\0';
    return true;
}
```

```
int main(int argc, char *argv[]) {
    uint8_t buffer[1024];
    CheckNumInput input = CheckNumInput_init_zero;
    pb_istream_t stream;

    // Setup string callbacks
    char operation_str[64];
    input.operation.funcs.decode = &string_callback;
    input.operation.arg = operation_str;

    // Read from stdin
    size_t bytes_read = fread(buffer, 1, sizeof(buffer), stdin);
    stream = pb_istream_from_buffer(buffer, bytes_read);

    // Decode Protocol Buffer
    if (!pb_decode(&stream, CheckNumInput_fields, &input)) {
        fprintf(stderr, "Decoding failed\\n");
        return 1;
    }

    // Call original function
    return original_checkNum(input.number, operation_str, input.verbose);
}
```