INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

Automated Task Generation for the Verification of C Programs

Integration of Verification-Task Generation into the C Build Process

Ibrahim Durmuscelebi

Bachelor Thesis

Supervisor	Dr. Thomas Lemberger
Submission Date	20.04.2025

Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. I used ChatGPT to generate and improve wordings of single sentences and small paragraphs. I used GitHub Copilot to suggest small snippets of code.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe ChatGPT genutzt, um Formulierungen einzelner Sätze und kleiner Absätze zu erstellen und zu verbessern. GitHub Copilot habe ich verwendet, um kleine Code-Snippets vorschlagen zu lassen.

München, 20.04.2025 Ibrahim Durmuscelebi

Abstract

Verification tasks for software analyzers typically require a single-file input that includes all necessary headers and parameters. Currently, this preparation is often done manually, which is time-consuming and creates more errors. HarnessForge is a verification task generator, which partially addresses this challenge by automating the creation of single-file verification tasks from multi-file projects. However, in its current state, HarnessForge requires users to manually identify and incorporate compile-time parameters into the execution command, adding complexity and increasing the risk of misconfiguration. This thesis extends HarnessForge to also record and integrate compile-time parameters, ensuring that headers, arguments, and other parameters are included automatically. As a result, developers can more easily use state-of-the-art analysis tools on complex projects without manual setup, advancing the adoption and reliability of automated verification solutions. The enhanced version of HarnessForge is evaluated on three large-scale real-world projects, demonstrating its effectiveness and compatibility with complex software systems.

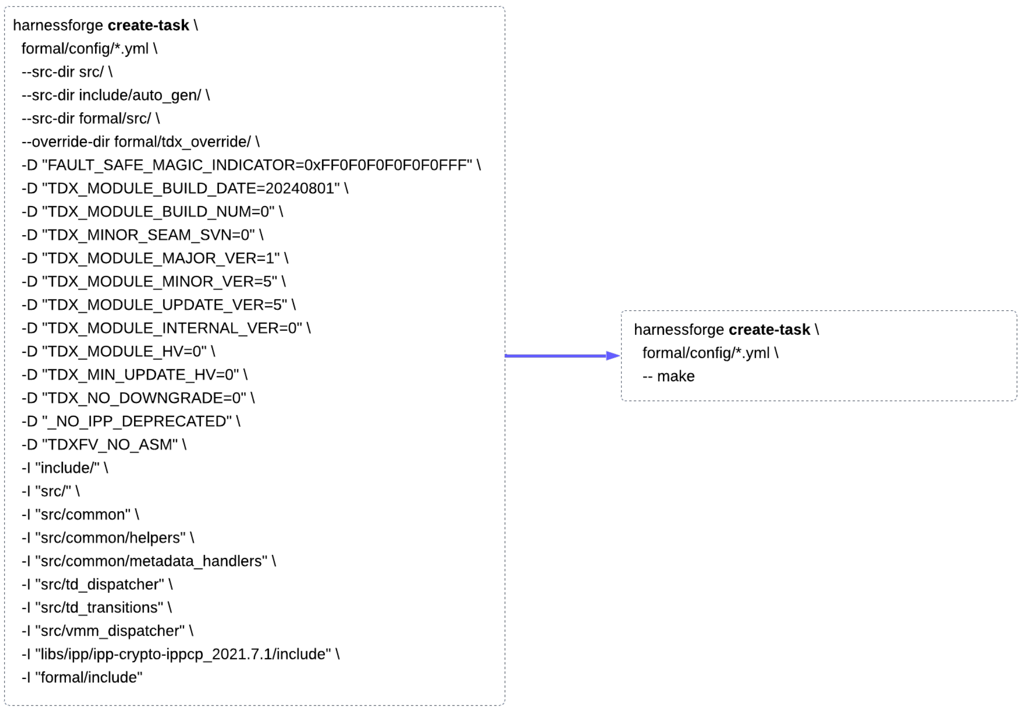
Contents

List of Figures

1 Introduction

1.1 Motivation

Testing software comprehensively to ensure the absence of bugs is a significant challenge, as software bugs are often considered inevitable. Software verification tools like HarnessForge aim to address this by providing automated model-checking and test-generation capabilities. However, the current implementation of HarnessForge requires significant manual input, such as specifying build parameters, source files, and include directories, which can introduce human errors into the verification process. **The goal** of this thesis is to **automate these manual steps**, enabling verification tasks to be generated with minimal user intervention, as illustrated in Figure 1.1. The left side of the figure highlights the complexity and verbosity of a typical manual command, requiring users to specify numerous directories, parameters, and macros individually. By contrast, the right side demonstrates the potential for simplification through automation, encapsulates the same functionality with significantly reduced effort. Beyond this, another key objective is the **evaluation and selection of tools** for recording build parameters, identifying the most suitable option for integration with HarnessForge. The selected tool will then be implemented and integrated into the system, ensuring compatibility with GNU Make and potentially other build systems. Finally, real-world validation will be conducted through case studies on large-scale projects, demonstrating the effectiveness and usability of the enhanced HarnessForge in handling complex verification tasks. By automating the discovery of dependencies such as headers, include files, and other configuration parameters directly from the source code, we can reduce the risk of human errors and streamline the verification process. Automation is particularly crucial for C programs, which are widely used in industrial applications where correctness is critical. Many C-based systems operate in safety-critical environments, such as automotive, medical, or aerospace industries, where software errors can have catastrophic consequences [5, 10, 13]. By improving HarnessForge's automation, this work contributes to making verification more robust and practical for such critical applications.



```
harnessforge create-task \
  formal/config/*.yml \
  --src-dir src/ \
  --src-dir include/auto_gen/ \
  --src-dir formal/src/ \
  --override-dir formal/tdx_override/ \
  -D "FAULT_SAFE_MAGIC_INDICATOR=0xFF0F0F0F0F0F0FFF" \
  -D "TDX_MODULE_BUILD_DATE=20240801" \
  -D "TDX_MODULE_BUILD_NUM=0" \
  -D "TDX_MINOR_SEAM_SVN=0" \
  -D "TDX_MODULE_MAJOR_VER=1" \
  -D "TDX_MODULE_MINOR_VER=5" \
  -D "TDX_MODULE_UPDATE_VER=5" \
  -D "TDX_MODULE_INTERNAL_VER=0" \
  -D "TDX_MODULE_HV=0" \
  -D "TDX_MIN_UPDATE_HV=0" \
  -D "TDX_NO_DOWNGRADE=0" \
  -D "_NO_IPP_DEPRECATED" \
  -D "TDXFV_NO_ASM" \
  -I "include/" \
  -I "src/" \
  -I "src/common" \
  -I "src/common/helpers" \
  -I "src/common/metadata_handlers" \
  -I "src/td_dispatcher" \
  -I "src/td_transitions" \
  -I "src/vmm_dispatcher" \
  -I "libs/ipp/ipp-crypto-ippcp_2021.7.1/include" \
  -I "formal/include"
```

```
harnessforge create-task \
  formal/config/*.yml \
  -- make
```

Figure 1.1: Comparison of verbose and minimized Intel-TDX verification commands

1.2 Mitigating Manual Input

The primary objective of this thesis is to integrate automation into the HarnessForge toolchain. The process begins with the identification and extraction of all necessary build parameters, such as header files, include directories, macros, and other compile-time elements from the C source files under verification. Existing mechanisms, such as the use of `-MJ` arguments in the C programming language, are explored to achieve this. Instead of executing the entire build process at once, we first run the `-MJ` command, which generates a JSON compilation database containing essential build parameters. This database includes information about the -I (include directories) and -D (macro definitions) parameters, ensuring that all necessary dependencies and configurations are captured. Once this database is created, the actual HarnessForge process is executed, extracting the required information directly from the JSON compilation database rather than relying on manual input. This two-step approach allows for greater flexibility, improves automation, and minimizes the risk of missing or incorrectly specified parameters Figure 1.2. The initial step involves parsing the build process to identify all dependencies and configurations required for verification. This includes headers, macros, and dynamic variables. For macros or variables that are predefined, the system prompts the user to confirm or modify their values as needed to ensure flexibility and accuracy. Once all dependencies are identified, the extracted information is structured into a JSON format compatible with HarnessForge. However, rather than being directly used for verification, this

JSON data is utilized to generate a **project-config.yaml** file. This configuration file encapsulates all necessary parameters for the verification process. The HarnessForge verification task is then executed using this **project-config.yaml**, rather than directly from the database. Additionally, the **project-config.yaml** provides three options for handling the extracted build parameters: (E) Edit, (U) Use, and (D) Dismiss. These options allow users to either modify the configuration before execution, directly use it as generated, or discard it entirely. This intermediate step enhances modularity and ensures that the verification setup remains adaptable and reproducible Figure 1.3. The appropriate option is selected based on the mode in which HarnessForge is running, ensuring flexibility and control over the verification setup. More details on these modes and their functionality will be discussed in Section 3.3.

To achieve the automated extraction and organization of build parameters, this thesis evaluates and implements one of the following existing tools: Bear, One-Line-Scan, or CodeChecker.
Each of these tools offers unique capabilities:

- Bear[1] specializes in capturing build parameters through the interception of build commands, producing a JSON compilation database that is widely compatible with static analysis and verification tools. Its primary focus is on capturing build parameters without introducing additional functionalities, making it lightweight and efficient for this specific purpose. This aligns directly with the objective of extracting compile-time parameters to automate verification tasks.
- One-Line-Scan[2] is a versatile tool that hooks into the compilation process, wrapping calls to the compiler with other tools. It facilitates not only the extraction of build parameters but also enables projects to be compiled for fuzzing with AFL or for static code analysis with tools like CBMC. While it offers broader capabilities, its additional features introduce complexity that may be unnecessary for the sole purpose of extracting build parameters.
- CodeChecker[3] is an extensive analyzer tooling, defect database, and viewer extension for static and dynamic analysis tools. It supports multiple analyzers, provides web-based report storage, and includes command-line tools for comprehensive analysis management. Although CodeChecker can extract build parameters, its comprehensive nature and focus on defect management make it more complex and resource-intensive compared to Bear.

Among these, Bear is the most suitable candidate for integration, as it aligns directly with the goals of dependency discovery and JSON-compatible output generation without introducing unnecessary complexity.

Simplify Build Command

To simplify the process of defining verification tasks, the command structure was extended to include a -- flag after the configuration file. This flag, along with the

[1] https://github.com/rizsotto/Bear
[2] https://github.com/awslabs/one-line-scan
[3] https://github.com/Ericsson/codechecker

make command, serves to streamline the process by referencing the configuration file containing the details of the target C file, associated methods, and other parameters. The code dynamically checks for the presence of the -- flag. If detected, the system verifies if additional compile arguments are provided. All additional parameters will also be added to the command after make This mechanism enables seamless integration with the Bear library for the next step. By automating and unifying these steps, the thesis achieves its primary goal of simplifying the preparation of verification tasks. The exact command structure of the goal, as implemented:

```
harnessforge create-task <config-file> <options> --
<build command>
```

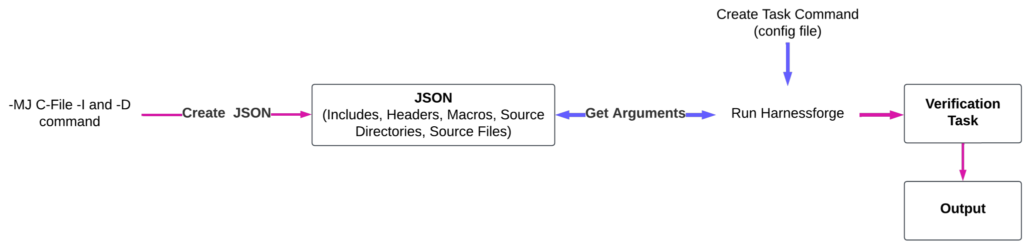


Figure 1.2: Example of an overview of the initial approach using `-MJ` arguments

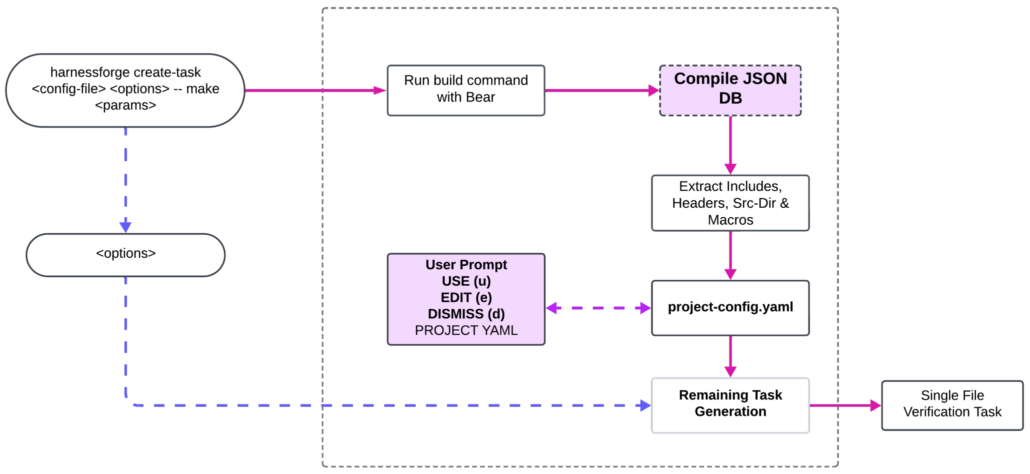

harnessforge create-task <config-file> <options> -- make <params>
<options>
Run build command with Bear
Compile JSON DB
Extract Includes, Headers, Src-Dir & Macros
User Prompt
USE (u)
EDIT (e)
DISMISS (d)
PROJECT YAML
project-config.yaml
Remaining Task Generation
Single File Verification Task

Figure 1.3: Example of an overview of the final goal approach

1.3 Scope

The scope of this thesis focuses on automating the process of preparing verification tasks in HarnessForge for C programs. The main objective is to integrate the necessary features so that all required arguments and configurations (e.g., `sources`, `includes`, and other C-specific arguments) are automatically identified and applied, enabling HarnessForge to execute verification with a single command.

1.3.1 Features to Implement

- Automate the extraction of required dependencies, such as headers, macros, and include directories, from the source code.
- Implement a mechanism to detect and record compile-time arguments using methods like `-MJ`, or by leveraging existing tools such as Bear, One-Line-Scan, or CodeChecker for build parameter extraction.
- Automatically structure the parameters into a JSON format and integrate them with HarnessForge.
- Ensure that the system handles interactive prompts for dynamic variables (e.g., macro overrides) where necessary or automatically add a value to those macros.
- Write Test's like multifile Projects (e.g., Intel TDX) with Makefiles.
- Implement a feature that allows the tool to run in a non-interactive mode `--non-interact` within CI pipelines. When this mode is enabled, the tool should execute without prompting the user for any input, using default configurations. If `--non-interact` is not specified, the tool should run in its default mode, prompting the user to review and modify the project-config.yaml file, not only for macros but for any configurable parameters before proceeding with the verification task.

1.3.2 Features to Ignore

- **Performance Optimization**: Initially, the focus is on ensuring correctness and usability, not on optimizing runtime performance. A delay of a few seconds is acceptable as long as the verification process works reliably.
- **Non-C Languages**: The thesis specifically targets C programs, excluding adaptations for other programming languages.
- **Advanced User Interface**: The focus is on command-line functionality rather than developing a graphical interface.

1.4 Internal Tests

To support the verification of complex, multi-file projects (e.g., Intel TDX) using Makefiles, a structured testing methodology is established. The goal was to assess how well HarnessForge could handle multi-file C programs while ensuring compatibility with real-world build systems. A dedicated project and test structure was designed, incorporating essential components such as header files, a main function, and configurable compilation flags. This structure allowed for modular testing, ensuring that dependencies between files were properly captured. The test setup involved defining and including header files, compiling the project with Makefiles, and leveraging the `-MJ` option to extract build parameters. The detailed project structure of this internal test is presented in Figure 1.4.

1.4.1 Initial Testing and Build Parameter Extraction

The first phase of testing focused on determining whether `-MJ` could effectively capture the necessary compilation metadata. The test cases were designed to simulate real-world scenarios where conditional compilation flags and cross-file dependencies play a crucial role. The use of macros (ONE_FLAG) within the tests allowed validation of dynamic compilation settings. After verifying the feasibility of `-MJ`, we introduced Bear, an essential tool for extracting compilation commands in more complex environments. Bear was employed to generate compilation databases, which provided a structured way to collect all necessary build parameters. This step was critical, as real-world projects often involve intricate build systems that rely on multiple dependencies and dynamic configurations.

1.4.2 Test Coverage and Configuration

The created test consisted of a C program with an associated configuration file to ensure that all relevant parameters were correctly captured within the HarnessForge workflow. The successful execution of this test confirmed that multi-file project verification with Makefiles could be integrated into the automated task generation pipeline. With these insights, the methodology was refined and extended, ensuring that the verification approach remains scalable and adaptable for more complex real-world projects. The results validated the feasibility of using HarnessForge for multi-file program analysis while maintaining compatibility with various build environments.

```
test_root/
├── configs/
│   └── test_config.yaml
├── src/
│   └── test.c
├── include/
│   └── header.h
└── Makefile
```

Figure 1.4: Project Structure for Multi-File Testing

1.5 Evaluation

1. Can the required dependencies (headers, macros, and compile-time arguments) be accurately and automatically extracted from C source files using the `-MJ` argument or an equivalent method?
2. Does the automated process reliably create a verification task that works without manual input for a wide range of real-world C projects?
3. Can the enhanced HarnessForge successfully verify software in environments supported by Intel TDX, Coreutils and AWS C Commons?
4. Are we obtaining the same compile-time parameters and results with a different build system (e.g., Make, Ninja)?

1.6 Related Work

In the domain of automated software verification for C programs, several tools and frameworks have been developed to enhance code reliability and detect potential errors. This chapter discusses notable tools and methodologies, highlighting their functionalities and distinguishing them from the approach proposed in this thesis.

CBMC and Klever: Manual Verification Challenges for Linux

CBMC [4] is a prominent tool that verifies C programs by checking for violations of specified assertions. It translates the program into a formula, enabling the detection of errors such as buffer overflows and pointer safety violations. CBMC is recognized for its precision and robustness, and it has been applied to real-world software, including parts of the Linux kernel [11].
Klever is designed to facilitate the application of automatic software verification tools to large-scale industrial C programs. It addresses challenges such as environment modeling, specification of requirements, and verification of multiple versions and configurations. Klever aims to reduce the effort required for verifying critical software components in industries like operating systems and embedded systems [13].
However both Systems needs a lot of human input and neither tool automates the discovery of compile-time parameters. This manual setup is time-consuming and error-prone.

CBMC in AWS: Scaling Formal Verification

AWS data centers rely on boot code as the first code executed, making it difficult to test due to its low-level nature and hardware dependencies [7]. CBMC has been applied to address these challenges, providing formal verification techniques to ensure memory safety and correct execution of such critical low-level code [7, 11].
AWS has extended its use of CBMC beyond boot code verification, integrating formal verification into its broader development workflows. AWS has developed a systematic methodology for applying symbolic model checking to a wide range of C-based systems, including custom hypervisors, encryption libraries, boot loaders, and IoT operating systems [6].

SV-COMP: Competition on Software Verification

The Competition on Software Verification (SV-COMP) is an annual event that evaluates and compares the performance of software verification tools. It provides a comprehensive benchmark suite for C and Java programs, facilitating the assessment of tool capabilities in detecting errors and verifying program properties. SV-COMP serves as a valuable resource for understanding the state-of-the-art in software verification [2]. One of the most prominent tools developed and extensively evaluated within the SV-COMP community is CPAchecker, a configurable software verification

[4] https://github.com/diffblue/cbmc

framework. The recent release of CPAchecker 3.0 highlights significant advancements in configurable program analysis, focusing on precision and scalability for industrial-scale verification tasks [1].
In addition, Beyer and Lemberger's comparative study, "Software Verification: Testing vs. Model Checking," presents a comprehensive evaluation of the strengths and limitations of testing and model checking techniques in software verification [3]. This work underscores the importance of combining multiple approaches to achieve thorough verification results, particularly in complex or safety-critical systems.

Moving Fast with Software Verification at Facebook

Facebook's Infer tool demonstrates how static analysis can be integrated into fast-paced software development workflows. Designed to detect issues such as null pointer exceptions and resource leaks, Infer enables ongoing verification as part of continuous integration pipelines. It operates by analyzing code changes incrementally, allowing developers to catch potential defects early in the development process without significantly impacting build times. This incremental analysis reduces the overhead of full program verification while ensuring that critical bugs are identified before deployment. This approach illustrates how verification can be automated and scaled effectively, aligning with the goals of these thesis. [4]

MAGIC: Efficient Verification of Sequential and Concurrent C Programs

Modular **A**nalysis of pro**G**rams **I**n **C** [5] tool applies a structured approach to software verification by simplifying complex programs into smaller, more manageable representations. It achieves this by using predicate abstraction, a technique that replaces detailed program states with abstract logical expressions that capture essential program behaviors [9]. Additionally, it refines these abstractions incrementally based on verification feedback, ensuring that only relevant program details are considered [5]. MAGIC focuses on dynamically refining predicates to improve verification efficiency, whereas this work concentrates on automating the extraction and integration of compile-time parameters to streamline verification task generation. Both approaches contribute to reducing manual effort and increasing scalability in software verification but address distinct aspects of the process—MAGIC at the abstraction refinement level and this thesis at the verification setup level.

Frama-C: Modular Verification through Plug-in Architecture

Frama-C [6] is a comprehensive source code analysis platform designed for the verification of industrial-scale C programs. It provides a suite of plug-ins that facilitate static analysis, deductive verification, and testing, particularly for safety- and security-critical software. The platform enables collaborative verification by integrating these plug-ins atop a shared kernel and data structures, all adhering to a common specification language [10].

[5] https://www.cs.cmu.edu/~chaki/magic/
[6] https://frama-c.com

The extensibility of Frama-C allows for various verification strategies to be applied based on the specific needs of a project. Its modular nature supports a combination of formal methods, enabling integration with different verification frameworks.

2 Background

The development and verification of C programs demand efficient compilation, building, and automation processes. These tasks are critical for ensuring correctness and consistency. As this thesis focuses on automating the build parameters for HarnessForge, it requires an integrated understanding of compilation, building, and automation. Together, these processes form the foundation for **automated task generation for the verification of C Programs**.

2.1 Compile with Clang

Clang is a modern compiler for the C family of programming languages, developed as part of the LLVM project [1]. Renowned for its modular design, fast compilation, and detailed diagnostics, Clang is designed to support modern C standards while producing high-quality intermediate representations (LLVM IR) [14]. These IRs form a program for analysis, enabling advanced optimization techniques and precise transformations, which are integral to tasks like code certification and verification. By leveraging LLVM's infrastructure, Clang provides developers with a powerful and extensible toolchain for building, analyzing, and certifying C programs.
The following example illustrates a typical use of the Clang compiler to compile a C source file:

```
clang -I <include_dir> -D <macro_definition> -o <output_file>
<source_file>
```

In this general command:

- `-I <include_dir>` specifies the directory containing header files to include during compilation.
- `-D <macro_definition>` defines a preprocessor macro to be used in the compilation process.
- `-o <output_file>` specifies the name of the output file (e.g., an executable or object file).
- `<source_file>` represents the C source file to be compiled.

[1] https://llvm.org/

With `-MJ` flag

As the main goal of this thesis is to build the compile commands in JSON format, Clang provides the `-MJ` flag for this purpose. The `-MJ` flag allows for automatic generation of a compilation database entry during the build process. This eliminates the need for manual entry of compilation parameters into a JSON file, ensuring accuracy and efficiency.
The following example demonstrates how Clang can be used to generate a JSON entry for a compilation database during the build process with `-MJ` flag:

```
clang -MJ <output_json> -I <include_dir> -D <macro_definition>
-o <output_file> <source_file>
```

This command is similar to the general compilation command described earlier, but with the addition of the `-MJ <output_json>` flag, which specifies the name of the JSON file where the compilation entry will be stored. The resulting JSON file might look as follows:

```
{
    "directory": "harnessforge/test/tasks/with-header",
    "file": "src/header.c",
    "output": "/var/folders/b0/zr4pr31j7c14rf/T/header-50b165.o",
    "arguments": [
      "/opt/homebrew/Cellar/llvm@15/15.0.7/bin/clang-15",
      "-xc",
      "--sysroot=/Library/Developer/CommandLineTools/SDKs/
      MacOSX14.sdk",
      "src/header.c",
      "-o",
      "/var/folders/b0/zr4pr31j7c14rf/T/header-50b165.o",
      "-I",
      "includes",
      "-D",
      "ONE_FLAG=200",
      "-mlinker-version=1022.1",
      "-mmacos-version-min=14.0.0",
      "-stdlib=libc++",
      "--target=arm64-apple-macosx14.0.0"
    ]
  }
```

The JSON file contains all necessary parts of the build command, including directories, macros, source files, and flags. Although we would get all the parameters in the JSON file, we still have to enter the build parameters in the `-MJ` command manually, which shows us the necessity of using an automated build parameter extraction tool.

2.2 Build with Make

Build tools are essential for transforming source code into executable programs, automating tasks such as compiling code, linking libraries, and packaging binaries. One of the most widely used build automation tools in C programming is Make,

[8] which simplifies dependency management and streamlines the generation of executables. Make operates through the use of a **Makefile**, a configuration file that defines the rules and dependencies required for building and linking the project [12]. This approach allows developers to handle complex build workflows in an efficient and organized manner, reducing manual effort and ensuring reproducibility. Below is an example Makefile used for the build process:

```
# Compiler and Flags
CC = clang
CFLAGS = -Wall -Wextra -D ONE_FLAG=200 -I includes

# Target executable
TARGET = header_test

# Source files
SRCS = src/header.c
OBJS = $(SRCS:.c=.o)

# Dependency files
DEPS = $(OBJS:.o=.d)

src/header.o:  src/header.c
# Build the target
all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Generate object files and dependency files
%.o: %.c
    $(CC) $(CFLAGS) -MMD -MP -c $< -o $@

# Include dependency files
-include $(DEPS)

# Clean up build artifacts
clean:
    rm -f $(TARGET) $(OBJS) $(DEPS)
```

The Makefile includes all the necessary build parameters, such as include directories, macro definitions, source files, compiler flags, and dependency rules, which are essential for correctly building a project. It automates the process of compiling source files into object files, linking them into an executable, and generating dependency files to manage build relationships efficiently. Additionally, the Makefile contains a clean target, which simplifies the removal of build artifacts, ensuring a clean and reproducible build environment. Its flexibility allows for easy modifications and extensions, such as adding support for debug or release builds, customizing compiler flags, or incorporating additional tools like linters or formatters.

2.3 Building Compilation Databases with Bear

Bear is an open-source tool designed to simplify the generation of compilation databases, which are critical for tools that analyze or manipulate code [2]. By inter-

[2] https://github.com/rizsotto/Bear

cepting the build process, Bear automates the creation of a JSON file, eliminating the need for manual configuration and ensuring compatibility with Clang. Bear works seamlessly with build systems like Make. When paired with Make, Bear captures all the commands, includes, and flags used during the build process, ensuring a complete and accurate representation of the compilation workflow. This makes it more convenient compared to the `-MJ` flag, which requires manual configuration and separate execution for each source file. The use of Bear is especially beneficial for this thesis, as it automates the generation of build parameters required by HarnessForge. By reducing manual effort and minimizing errors, Bear streamlines the verification process and ensures consistency in build workflows. Using Bear with Make significantly simplifies the build command. Instead of specifying individual commands and parameters:

```
bear -- make
```

JSON file generated by Bear:

```
    [
        {
          "arguments": [
            "/opt/homebrew/opt/llvm@15/bin/clang",
            "-Wall",
            "-Wextra",
            "-D",
            "ONE_FLAG=100",
            "-I",
            "includes",
            "-c",
            "-o",
            "src/header.o",
            "src/header.c"
          ],
          "directory": "harnessforge/test/tasks/with-header",
          "file": "harnessforge/test/tasks/with-header/src/header.c",
          "output": "harnessforge/test/tasks/with-header/src/header.o"
        }
      ]
```

The JSON output is nearly identical to what is generated using the `-MJ` flag, with slight differences. However, the key build parameters, such as includes, flags, and source files, remain the same. This consistency allows seamless integration with HarnessForge, enabling easier and more automated verification processes without requiring manual input.

At the end of this section, a figure illustrates the connection and process flow between bear -- make, compilation, and building, showcasing how these elements work together Figure 2.1.

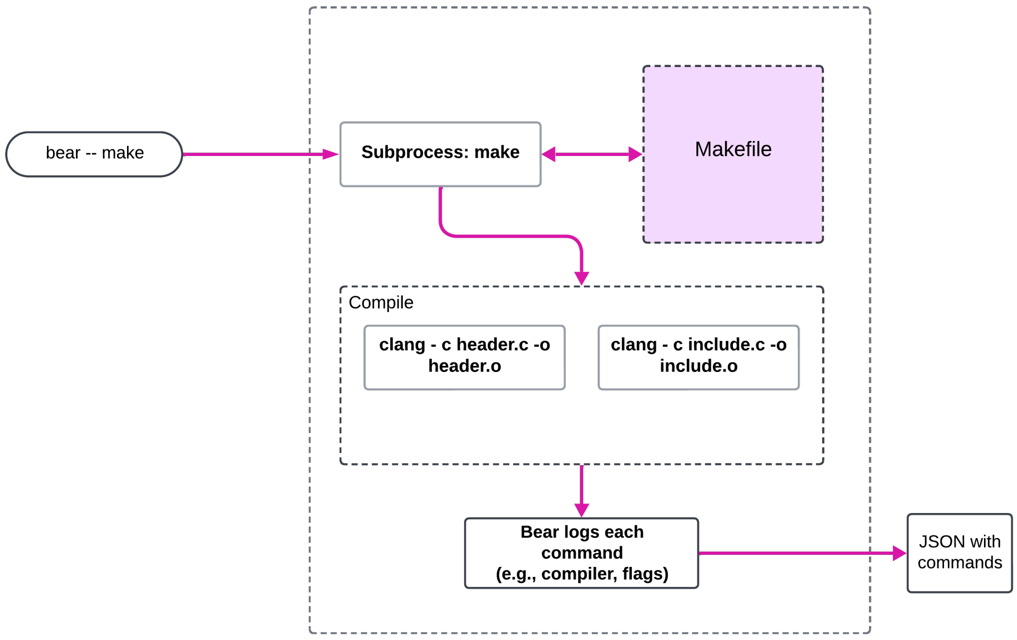

bear -- make
Subprocess: make
Makefile
Compile
clang - c header.c -o header.o
clang - c include.c -o include.o
Bear logs each command (e.g., compiler, flags)
JSON with commands

Figure 2.1: Process Flow for `bear -- make` with Makefile, header.c and include.c

3 Contribution

The primary contribution of this thesis is the enhancement of HarnessForge to fully automate the generation of verification tasks for C programs, eliminating the need for manual configuration of compile-time parameters, as illustrated in Figure 1.1. This section provides a detailed explanation of the steps and methods undertaken to achieve this objective and emphasizing the practical implementation. The exact command structure of the implemented solution is as follows:

```
harnessforge create-task <config-file> <options> --
<build command>
```

3.1 Bear Integration

Bear operates as an external process within the code. The command used to invoke Bear is as follows:

```
bear --output <json-file> -- <build command>
```

In this command, `bear` is invoked with the `-output` option, specifying a temporary JSON file (`json_file`) to store the compilation details. The `--` flag separates Bear-specific options from the subsequent **build tool** and its command, which is defined by the user along with any additional compile-time arguments, as specified in the original HarnessForge command. The external process workflow simultaneously builds the C project and generates the JSON file through Bear. During this process, Bear intercepts the build execution by wrapping the `make` command and capturing all relevant details, including source file paths, include directories, macros, and compiler flags. It is also ensuring that the build is executed correctly and without errors, guaranteeing the accuracy and reliability of the generated JSON file. The resulting JSON file, created as a temporary artifact, serves as an essential input for preparing verification tasks.

3.2 Extract JSON

The next step in the workflow involves processing the JSON file generated during the build process to extract critical compilation parameters, such as include directories and macro definitions. These parameters are essential for accurately recreating the original build environment within the verification tasks. The extracted data is structured into a `NamedTuple` and subsequently passed to the verification task generator. To ensure the JSON file is correctly parsed, trailing commas are removed from its content. This precaution is necessary because the `-MJ` flag in Clang sometimes appends a trailing comma in the JSON output, which can result in invalid JSON formatting. By addressing this issue, the system ensures robust handling of the generated data. For multi-file projects, the JSON file can contain numerous entries, each corresponding to a separate compilation command. To manage this complexity, additional checks ensure that duplicate entries, such as `-I` include paths or `-D` macros, are not redundantly added. This is particularly important because the same `-include` or `-macro` may be required for different files in a project, and adding duplicates could lead to unnecessary clutter in the JSON file or errors during processing. Additionally, the `source` file is included by checking each argument, where one of them corresponds to the `.c` file. Since the file is explicitly added as `sources`, it ensures correct handling of source file locations within the verification setup. Additionally, the system accounts for variations in how arguments are provided, such as `"-I src/includes"` versus `"-Isrc/includes"` or similar patterns for macros. Both formats are checked and processed to ensure that all relevant values are correctly captured and included in the final verification task preparation. This flexibility guarantees compatibility with various argument styles, ensuring the JSON database reflects the actual build configuration accurately. For example, in the Intel-TDX project, the JSON output exceeds approximatly 50,000 lines. To illustrate this complexity, just the first two arguments from the JSON file might look like:

```
{
    "arguments": [
      "/usr/bin/clang-15",
      "-c",
      "CMakeCCompilerId.c"
    ],
    "directory":
        "/mnt/macfiles/intel-tdx/libs/ipp/ipp-crypto-ippcp_2021.7.1/
    _build/CMakeFiles/3.28.3/CompilerIdC",
    "file":
        "/mnt/macfiles/intel-tdx/libs/ipp/ipp-crypto-ippcp_2021.7.1/
    _build/CMakeFiles/3.28.3/CompilerIdC/CMakeCCompilerId.c"
  },
  {
    "arguments": [
      "/usr/bin/clang++-15",
      "-c",
      "CMakeCXXCompilerId.cpp"
    ],
    "directory":
        "/mnt/macfiles/intel-tdx/libs/ipp/ipp-crypto-ippcp_2021.7.1/
    _build/CMakeFiles/3.28.3/CompilerIdCXX",
    "file":
        "/mnt/macfiles/intel-tdx/libs/ipp/ipp-crypto-ippcp_2021.7.1/
    _build/CMakeFiles/3.28.3/CompilerIdCXX/CMakeCXXCompilerId.cpp"
  },
```

Such large files necessitate iterating through all arguments to ensure that every relevant include path and macro definition is captured. This approach ensures that even in projects with extensive build environments, the workflow reliably extracts the required information.This process not only eliminates manual effort but also minimizes the risk of errors, ensuring that verification tasks reflect the actual build context, thus enhancing the reliability of the analysis.

3.3 Project Config File YAML

Once the necessary compilation parameters have been extracted from the `JSON` file, the next step involves generating a **project-config.yaml** file. This YAML configuration file serves as an intermediate representation that encapsulates all essential parameters required for the verification process. Rather than directly using the extracted JSON data, the system translates it into a structured YAML format, ensuring compatibility with **HarnessForge** while maintaining modularity and reusability.

YAML instead of JSON?

Although the extracted compilation database is stored in `JSON` format, `YAML` is preferred for configuring verification tasks due to its improved readability, human-editability, and flexibility in defining structured configurations. YAML files allow for better organization of nested settings and enable additional manual modifications when necessary. For further details of the manual interaction, see: Non Interactive.

Ensuring correctness

To ensure correctness, a **YAML preview step** is introduced, allowing the user to inspect the generated configuration. The user is presented with the newly created `project-config.yaml` and has the following options:

- **Use (u)** – Accept the file and proceed with the verification task.
- **Edit (e)** – Modify the YAML file before execution.
- **Dismiss (d)** – Reject the new file and use the previously existing configuration, if available.

The following preview demonstrates how this step appears:

```
Preview of the YAML that will be created:

sources:
- src/with-header.c
source_dirs:
- src
includes:
- includes
defs:
- ONE_FLAG: 100
override_dir: ''

Do you want to use (u) this YAML, edit (e) it, or dismiss to
```

```
use existing config (d)?
```

This mechanism is essential because the `project-config.yaml` file may be empty if a new build is not created or if an existing build is reused (object, dependency files not cleaned), potentially leading to issues. Rather than enforcing automatic cleanup of old builds, the approach ensures greater flexibility by allowing users to dynamically manage configurations before verification begins. Through the YAML preview step, users can inspect the generated configuration and make informed decisions. If the file is empty or incorrect, the `edit` option allows for modifications, while `dismiss` enables the use of a previously existing configuration. This process ensures that users remain fully aware of the state of the configuration file and have complete control before proceeding with the verification task.

3.4 Features

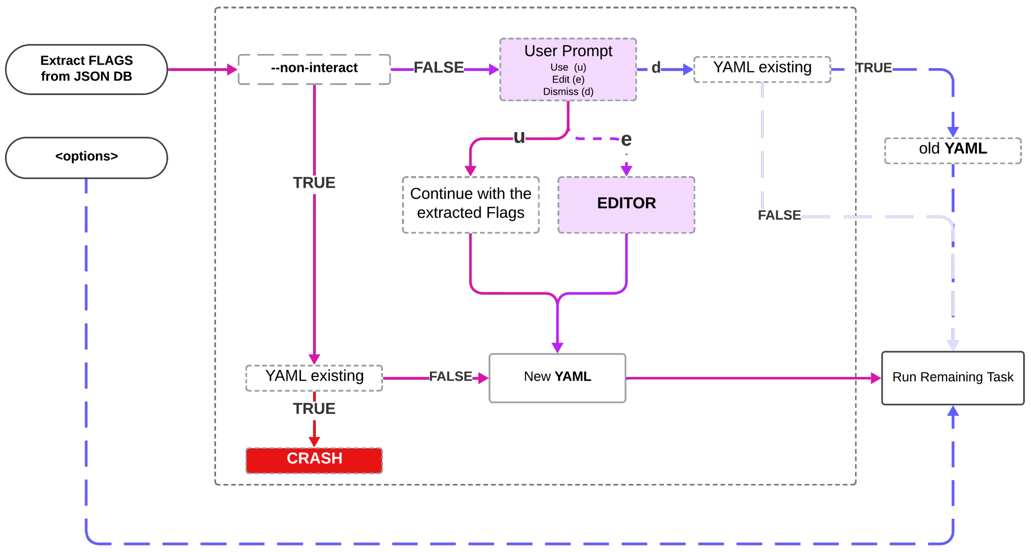


Figure 3.1: System workflow demonstrating the impact of the `--non-interact` option after JSON database extraction.

To enhance the usability and functionality of HarnessForge, several new features were added to make the tool more efficient, user-friendly, and adaptable for different use cases While the focus of this thesis is on ensuring correctness and reliability for C program verification tasks, features such as performance optimization, support for non-C languages, and advanced user interfaces were deliberately excluded to maintain this focus.

3.4.1 Non-Interactive Mode

A significant addition is the `--non-interact` option, which enables a fully automated execution mode, eliminating the need for user interaction. This feature is particularly beneficial for integration into automated workflows, such as continuous integration (CI) pipelines, where manual input is neither feasible nor desirable. To ensure correctness and avoid inconsistencies, non-interactive mode enforces strict handling of the `project-config.yaml` file. Before a new configuration file is created, a check is performed to determine whether an existing version is present. If a pre-existing `project-config.yaml` file is detected, execution is immediately aborted, and an error is raised. This precaution prevents outdated configurations from being inadvertently reused, ensuring that each verification task starts with an accurate and up-to-date setup. If no prior configuration file is found, a new `project-config.yaml` is automatically generated using the extracted parameters from the `JSON` database. This guarantees that the verification task is always executed with a clean and reproducible configuration, preventing potential mismatches between previous builds and the current analysis. By strictly enforcing this approach, non-interactive mode ensures reliability and consistency in verification workflows. Figure 3.1, illustrates how configuration files are handled in non-interactive mode.

4 Evaluation

In this section, the enhanced HarnessForge is evaluated based on its ability to extract compile-time parameters and generate verification tasks automatically. The evaluation consists of two main parts: First, tests are conducted using simplified default parameters and GNU Make to verify the correctness and completeness of the extracted dependencies. Second, real-world projects are analyzed to assess the tool's robustness and adaptability across different build systems, including CMake and Ninja. The primary focus is on determining whether the verification tasks generated by HarnessForge can be executed without manual intervention. We deliberately ignore detailed performance metrics such as execution time or CPU usage. Instead, the emphasis lies on functional correctness and automation.

To conduct these experiments, an Ubuntu environment (24.04.1 LTS) [1] is used with x86_64 architecture running on Docker 29.0 and 16GB of RAM. This setup is chosen because, running the experiments in a controlled Linux-based containerized environment ensures consistency and minimizes issues related to platform-dependent dependencies. Additionally, the following tools were used:

Tool	Version	
Bear	3.1.5	https://github.com/rizsotto/Bear
GNU Make	4.3	https://www.gnu.org/software/make
CMake	3.28.3	https://cmake.org

Tools used for evaluation

4.1 Intel-TDX

The Intel TDX project [2] is a large-scale, security-critical codebase developed to support Intel's Trusted Domain Extensions (TDX) technology. It provides infrastructure for secure virtual machine isolation by enabling hardware-enforced trusted execution environments. The project includes a wide range of low-level C components, such as boot logic, memory management, and cryptographic primitives, and spans hundreds of source files with complex hardware interactions. Building and

[1] https://www.ubuntu.com/

[2] https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html

compiling such a project is non-trivial, as it typically requires carefully configured environments, platform-specific dependencies, and custom toolchain setups—making it difficult to replicate or verify outside of its intended development context. It's size and complexity make it an ideal target for evaluating verification tools on realistic, production-grade systems[3]. Already successfully tested with HarnessForge, the Intel TDX project is revisited in this evaluation to assess the effectiveness of the automated parameter extraction process and since the necessary configurations for predefined methods are available, this project serves as a benchmark for testing the automated parameter process. This benchmark is particularly important because running Bear on the Intel TDX build generated a compile database file containing over 57,000 lines, including numerous repeated -D and -I flags. Manually identifying the relevant arguments for HarnessForge from this massive and redundant output would be highly time-consuming and error-prone. By contrast, the automated parameter extraction process integrated into HarnessForge is able to efficiently filter out duplicates and extract only the essential arguments, significantly accelerating the setup of verification tasks and ensuring correctness with minimal manual effort.

Results

To keep the evaluation concise, the full output is not included here due to its size. The automated process generated significantly more information than the manually specified parameters. This was expected, as the extraction directly lists individual source files rather than relying on broader source directory definitions. Additionally, in contrast to the manual setup, the automatically extracted configuration included 130 macro definitions beginning with `__FILENAME__`, which were not required for verification and had to be removed manually. The configuration also included irrelevant system directories, such as `/usr/cmake/modules` and `x11`, which were likewise removed to avoid conflicts during verification. Although the extracted configuration was more extensive and required manual refinement, the edit functionality of HarnessForge's `project-config.yaml` feature enabled easy cleanup of unneeded entries. After removing these unnecessary components, the resulting configuration performed as expected, and the verification process completed successfully. This highlights the value of the editing step in managing large-scale projects, where the initial compile database can exceed 57,000 lines and still result in over 1,000 lines after extraction and deduplication—making selective manual cleanup an essential part of the workflow.

4.2 Coreutils

GNU Coreutils [4] is a collection of fundamental command-line utilities for Unix-like operating systems. These tools provide essential functionalities for file manipulation, text processing, and system management, including commands such as `ls`, `cp`, `mv`, and `rm`. One of the utilities in Coreutils is `cat`, which is used to read and concatenate files. The `cat` command prints the content of files to standard output and is

[3]https://github.com/intel/tdx-module

[4]https://www.gnu.org/software/coreutils/manual/coreutils.pdf

commonly used for viewing file contents, combining multiple files, or redirecting output. A basic example of using `cat` is:

```
echo "Hello, World!" > test.txt
cat test.txt
```

The first command creates a file named test.txt and writes the text "Hello, World!" into it. The second command (cat test.txt) attempts to display the contents of test.txt.

4.2.1 Building Coreutils

To ensure consistency with the Intel-TDX evaluation, Coreutils was built in the same controlled environment. Unlike typical software projects structured around a single entry point, Coreutils consists of numerous independent utilities, each containing its own `main()` function. The Makefile for this project is particularly large—around 28,000 lines—which made the build process complex and error-prone. Given the importance of Coreutils as a widely used system utility suite, extra care was taken to verify each build step to ensure correctness and reproducibility for later verification tasks. During the process, it is discovered that building the entire project is not necessary for our purposes. Since the evaluation focuses on the `cat.c` component, the command `make src/cat` is used to compile only the relevant parts of the project. This approach significantly simplifies the build process and ensures that only the necessary files and dependencies for `cat.c` are included.

4.2.2 Creating `task-config.yaml` for `cat.c`

After successfully building Coreutils, a task configuration file (`task-config.yaml`) is created specifically for the verification of the `cat.c` component. This configuration defines the verification setup by specifying the relevant source files, dependencies, and verification properties. In HarnessForge, each verification task is generated as a single file containing its own `main()` function. Due to this, the original `main()` method from `cat.c` cannot be directly used as the entry point for verification and the function name had to be changed (in our case: to `__main()`), as it would cause a conflict. Therefore, the verification setup explicitly defines an alternative starting point, selecting the function `next_line_num` in `cat.c`. Using this function as the entry point ensures the verification task remains conflict-free and correctly structured for HarnessForge's single-file verification process.

4.2.3 Baseline verification with manually configured parameters

To validate the correctness of the generated verification setup, the process is first executed using manually configured parameters. Identifying the correct parameters requires several manual steps, including inspecting the source files to determine which headers were included and locating the corresponding include directories. Additionally, it is necessary to identify all relevant macro definitions used in the code. This process is time-consuming and requires close attention to detail, as each missing include or macro definition would result in verification failure. However, through manual inspection and iterative refinement, all necessary parameters are eventually identified. Once the complete set of dependencies, macros, and source files are provided, the verification succeeded. This manually assembled configuration serves as the baseline for evaluating the effectiveness and completeness of the automated extraction process.

4.2.4 Automated Parameter Extraction

To evaluate the automated extraction capability, the verification process is repeated using the enhanced version of HarnessForge. Similar to previous evaluations, the build is performed using the command in Section 3, and the parameters are automatically captured by Bear and integrated into the generated `project-config.yaml`. This automation significantly accelerated the overall setup process, as it eliminated the need to manually inspect source files for include paths, macros, and dependencies. Instead of spending time navigating the codebase to collect the necessary parameters, the relevant data is extracted automatically and structured into a usable configuration file. In the case of Coreutils, Bear generates a compile database of approximately 18,000 lines—even though only the `cat.c` utility is built and evaluated. Considering that Coreutils consists of more than 100 similar utilities, a complete build would have resulted in a vastly larger and more complex compile database. The automated parameter extraction integrated into HarnessForge greatly simplifies this complexity by focusing on the specific build invocation and filtering out irrelevant data, making the verification setup more efficient and scalable.

Results

The results confirm that the automated extraction correctly identifies essential dependencies and configurations from the targeted source file. This verifies the practical benefit of performing verification tasks individually on self-contained programs, rather than on the entire source directory structure. Unlike the Intel-TDX evaluation, no unnecessary system directories are included in the configuration. However, a few irrelevant dependencies are still present. For example, both `base32.c` and `base64.c` are listed as source files, even though only one of them is typically used depending on the system configuration. These kind of redundant entries are manually removed during refinement. This issue arises because the compilation and the resulting compile database created by Bear are driven by the Makefile, which, as described above, spans over 28,000 lines. Due to its size and complexity, it is practically infeasible to trace exactly why certain files are included in the compile database. Nevertheless, the required adjustments in this evaluation are minor compared to the extensive clean-up required for Intel-TDX, demonstrating the relative simplicity and effectiveness of automated parameter extraction in modular, utility-based projects like Coreutils.

4.3 AWS C-Common

AWS C-Common[5] is a foundational C library developed by Amazon Web Services, providing low-level functionality such as memory management, string handling, linked lists, hash tables, and other utility functions. It plays a critical role in the AWS C SDK ecosystem by offering standardized, efficient, and reusable components that are shared across multiple AWS libraries and services. This modularity ensures consistency, performance, and maintainability in high-scale cloud environments. In this evaluation, the focus is placed on the `command_line_parser.c` file, which provides a utility for parsing command-line arguments in a safe and structured manner. It includes functionality to handle argument tokens, option parsing, error detection, and value extraction. This component is used in AWS projects that require reliable and predictable command-line input handling. Verifying this module is particularly useful for evaluating how well automated parameter extraction works on components that operate independently but still follow complex internal logic and platform abstractions. Its modular nature and integration into AWS tooling make it a representative candidate for testing real-world verification workflows with HarnessForge.

4.3.1 Building AWS C-Common

AWS C-Common is successfully built in the same controlled environment used for previous evaluations. Similar to the Coreutils project, attempts to verify the entire source directory resulted in errors due to conflicts and complexities. Therefore, verification is limited to individual source files, ensuring a targeted and conflict-free approach. Unlike standard C projects, the build setup of AWS C-Common introduces additional complexity. The project does not place its Makefiles and build configurations directly in the main source directory. Instead, it expects the build to be performed from a separate directory. This non-standard structure requires adjustments to the usual build and extraction process, making the setup more involved than typical flat-directory C projects. Despite this, it is possible to build the project using both the `make` and `ninja` build systems. To evaluate flexibility and compatibility, both build methods are tested. However, unlike in Coreutils, there is no interactive utility like `cat` to test post-build functionality. In this case, verification relies entirely on the assumption that the internal unit tests included in the build system passed and that the generated binaries are structurally valid. This makes the verification process more abstract and dependent on internal correctness rather than observable output behavior.

4.3.2 Manual Parameter Extraction

As described in the Coreutils: Baseline verification with manually configured parameters, manual parameter extraction for AWS C-Common also involves inspecting the source file to identify the required include paths and macro definitions. The necessary parameters are extracted by directly examining the dependencies within

[5] `https://github.com/awslabs/aws-c-common`

command_line_parser.c and ensuring that only relevant flags and paths are included. Unlike in Coreutils, however, there is no need to modify or bypass a main() function. Since AWS C-Common is structured as a utility library rather than a collection of executable programs, its source files are not organized around entry points. This simplified the manual verification setup slightly and reduced the need for structural workarounds.

4.3.3 Automated Parameter Extraction with **make**

For AWS C-Common, the full make build command is executed to capture the build parameters. The generated project-config.yaml configuration initially included not existing paths, such as bin/system_info/include, and all source files from the project. These unnecessary paths are manually removed, and the source files are refined to include only the specific targeted file (command_line_-parser.c) required for verification. After these adjustments, the configuration is successfully verified, accurately matching the manual baseline.
YAML file generated with the make build command:

```
sources:
- source/allocator.c
...
- bin/system_info/print_system_info.c
source_dirs:
- source
...
- tests
- build/tests
- tests/logging
- bin/system_info
includes:
- source/external/libcbor
- include
- build/generated/include
- tests
- bin/system_info/include
defs:
- __GCC_HAVE_DWARF2_CFI_ASM: 1
- AWS_AFFINITY_METHOD: AWS_AFFINITY_METHOD_PTHREAD_ATTR
- AWS_PTHREAD_GETNAME_TAKES_3ARGS: 1
...
override_dir: ''
```

4.3.4 Automated Parameter Extraction with **ninja**

Similarly, the ninja build system is utilized to evaluate automated parameter extraction. Like the make build, the configuration generated by ninja included all source files within the project and irrelevant paths, which had to be manually refined. After removing these unnecessary paths and specifying only the targeted source file, the refined configuration matched the manually configured baseline, confirming successful and accurate automated parameter extraction and verification using ninja . The only difference was that the parameters in the project-config.yaml were not in the same order, which was not important.

YAML file generated with the `ninja` build command:

```
sources:
- source/arch/intel/asm/cpuid.c
...
- bin/system_info/print_system_info.c
source_dirs:
- source
...
- tests
- build/tests
- tests/logging
- bin/system_info
includes:
- source/external/libcbor
- include
- build/generated/include
- tests
- bin/system_info/include
defs:
- AWS_AFFINITY_METHOD: AWS_AFFINITY_METHOD_PTHREAD_ATTR
- AWS_PTHREAD_GETNAME_TAKES_3ARGS: 1
...
override_dir: ''
```

5 Future Work

5.1 Support for Projects with Existing `main()` Functions

As demonstrated in the Coreutils evaluation, projects that define their own `main()` function cannot be directly verified using the current HarnessForge setup, which generates its own `main()` entry point as part of the task generation process. This results in compilation or linking conflicts, requiring manual intervention—such as selecting a different function as the entry point, modifying the source code, or refactoring verification targets. These workarounds are not ideal and reduce the level of automation HarnessForge aims to provide. Future work should focus on extending HarnessForge's capability to support real-world software projects with existing `main()` methods. Potential improvements include automatically renaming or wrapping existing entry points or modifying the task generation logic to dynamically integrate user-defined entry functions.

5.2 Filtering of System Paths and Libraries

During automated parameter extraction, especially with the `make` build system, system directories such as `/usr/include/x11`, `/usr/cmake/module`, and other unrelated system paths are often included in the generated configuration. These directories are usually unnecessary for the verification task and can introduce irrelevant, duplicated, or conflicting dependencies—leading to cluttered configurations and increased risk of verification errors. This issue was particularly noticeable in real world projects. Future improvements should aim to integrate a filtering mechanism into the extraction pipeline to automatically detect and exclude system paths and libraries that are not directly relevant to the target verification task. Such filtering could be based on project-specific include hierarchies, known system paths, or pattern matching for exclusion.

6 Conclusion

The automated task generation for the verification of C programs significantly improved the usability of HarnessForge and marks a major step toward fully automating verification setup for complex codebases. By removing the need for manual specification of compile-time parameters, the extended version of HarnessForge streamlines the process and reduces sources of human error.
This enhancement not only accelerates the verification of multi-file projects but also increases reliability by ensuring consistent and complete extraction of essential parameters. Overall, the automation introduced in this work contributes to making verification processes faster, more scalable, and better suited for real-world software systems.

Bibliography

[1] D. Baier, D. Beyer, P. Chien, M. Jakobs, M. Jankola, M. Kettl, N. Lee, T. Lemberger, M. L. Rosenfeld, H. Wachowitz, and P. Wendler. Software verification with cpachecker 3.0: Tutorial and user guide. In A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 543–570. Springer, 2024.

[2] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024.

[3] D. Beyer and T. Lemberger. Software verification: Testing vs. model checking - A comparative evaluation of the state of the art. In O. Strichman and R. Tzoref-Brill, editors, *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2017.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[5] S. Chaki, E. M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods Syst. Des.*, 25(2-3):129–166, 2004.

[6] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow at amazon web services. *Softw. Pract. Exp.*, 51(4):772–797, 2021.

[7] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model checking boot code from AWS data centers. *Formal Methods Syst. Des.*, 57(1):34–52, 2021.

[8] S. I. Feldman. Make-a program for maintaining computer programs. *Softw. Pract. Exp.*, 9(4):255–65, 1979.

[9] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 191–202. ACM, 2002.

[10] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects Comput.*, 27(3):573–609, 2015.

[11] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[12] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol. Make it simple: an empirical analysis of GNU make feature use in open source projects. In A. D. Lucia, C. Bird, and R. Oliveto, editors, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, pages 207–217. IEEE Computer Society, 2015.

[13] E. Novikov and I. S. Zakharov. Verification of operating system monolithic kernels without extensions. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 230–248. Springer, 2018.

[14] M. Schordan, D. Beyer, and I. Bojanova. Software verification tools (track introduction). In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 177–181. Springer, 2020.