# Logging - The Missing Component For GCC Plugin Testing

Nimantha Kariyakarawana
*imec-DistriNet*
*KU Leuven*
Leuven, Belgium
nimantha.kariyakarawana@kuleuven.be

Tom Holvoet
*imec-DistriNet*
*KU Leuven*
Leuven, Belgium
tom.holvoet@kuleuven.be

***Abstract*—The importance of GCC plugins cannot be overstated in today's development of complex, often embedded systems software. GCC plugins play a crucial role in e.g. Linux kernel development, static program analysis, and compiler extension development. Surprisingly, however, very little work has been devoted to rigorous testing of such plugins. Even more, apparent core features of logging - enabling to monitor the execution of plugin code are missing. In this position paper, we want to argue for the importance of logging as a core feature in plugin development, as a crucial instrument for testing and debugging, security analysis and optimisation. We also expose why logging is not straightforward and why typical logging frameworks cannot be used for GCC plugins. We then sketch how we intend to add such logging, exclusively using components and libraries accessible to the GCC plugin API.**

***Index Terms*—GCC, GCC Plugins, Logs, Testing, Compiler**

## I. Introduction

GNU Compiler Collection or, in its most known form, GCC is a compiler collection produced by the GNU project [1]. GCC plugins are part of GNU compiler collection [1]. GCC plugins play a significant role in present-day compiler development. They provide the ability for developers to introduce new features to the compiler without changing its core structure. Furthermore, they are improving efficiency and decreasing the time needed for building and testing new features. GCC plugins possess the characteristic that fosters an environment which allows experimentation [2]. Therefore, it allows the developers to potentially experiment with different compiler features before the feature could become part of GCC. An example of such experimentation is the case of embedded system development. When performance and hardware are considered, GCC plugins provide the ability to create custom compiler behaviour that suits the embedded systems. Moreover, GCC plugins have simplified the tasks of programmers by eliminating the need to delve into the GCC source code for modifications. Well-known use cases for GCC Plugins are run-time-instrumentation, static analysis, experimentation with compiler features, data gathering for program analysis, etc. [3]. Since its release, plugins have been widely developed and used, and are often publicly accessible. Some notable plugins are Dehydra [3], the Linux-Kernel GCC Plugin [3] and the GCC Python Plugin [3].

Due to its versatility, GCC has become one of the most used compilers for software projects. It supports the compilation of multiple programming languages. However, it is the most used compiler for C and C++ [1]. Even though it is a powerful compiler, it is not without its limitations [4]. Due to the evolving nature of the industry, out-of-the-box features provided by GCC were not always sufficient. Before GCC version 4.5, programmers had to figure out how to overcome the limitations. New features were added by making direct changes to the compiler source. This was not an easy operation because it required a thorough understanding of the compiler's source code [5]. Additionally, the programmers had to ensure the modifications had not compromised the default functionality of the compiler.

GCC introduced a plugin API in version 4.5. The plugin API allowed the writing of custom modules as extensions to the compiler. These modules, more commonly known as GCC plugins, allowed third-party programmers to extend the compiler's functionality without changing its source code [5]. Moreover, the plugin API enabled the ability to access GCC internals, including AST, GIMPLE, RTL, and more [6]. GCC plugins have their separate source and are written in C or C++. Plugins are compiled into a shared library accessible by the compiler [7]. GCC plugins execute at compiler run time instead of application run time. GCC plugins can influence the compilation process at different stages. Plugins can be invoked at multiple stages of the compilation process, such as parsing, optimisation, code generation, etc. This enabled the plugin to modify the code before generating the final executable [8]. In the absence of thorough verification and testing, GCC plugins can compromise GCC and its workflow. Traditional software testing frameworks focus on run-time verification. Therefore, the internal behaviour of GCC plugins cannot be tested with the traditional frameworks. This is due to the difference in execution environments which are application run-time and compiler run-time. Moreover, the information used by the software testing frameworks is mostly available during the application run-time. In addition to that GCC, internals cannot be mocked or created during application run-time. However,

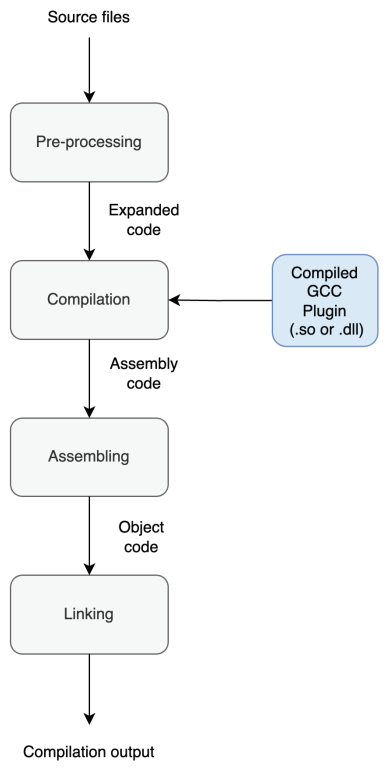

Fig. 1. Extended GCC Compilation process

the internal behaviour of the GCC plugins can be monitored with logs. They provide crucial insight into how GCC plugins behave internally.

Logging is the practice of recording information generated during the execution of a program. Log entries include information, event data, errors, messages, etc. Logs are crucial for debugging, security, testing, troubleshooting, etc. [9]. Logging functions as a popular technique to understand the plugin execution process internally. It is possible to monitor the execution flow and see the values at different stages of the compilation process by writing log statements. However, current logging frameworks are unusable. This is mainly because current frameworks are developed to test traditional C and C++ applications at run-time. Information used by traditional frameworks is not available through the GCC plugin API. Therefore, when GCC plugins are considered most of the logging has to be done manually. Currently, logging is achieved by writing print statements to the console or writing log statements to file [10]. We intend to share our insights with the research community by highlighting the significance and necessity of logging in GCC Plugins.

The rest of the paper is structured as follows. Section 2 presents the related work. Section 3 covers the discussion. Section 4 delves into the logging for GCC plugins. Section 5 discusses the conclusions & future work.

## II. Related work

GCC Plugins have been in existence since the release of GCC 4.5. However, the available academic studies on this domain are limited. Callanan et al. have developed a plugin that allows GCC to load GIMPLE transformations [11]. They employed logs based on GIMPLE that correspond to conditional statements, function calls, and variables. Furthermore, in their solution, it is possible to configure which GIMPLE-related information is logged. However, their focus is limited to GIMPLE [11]. Kumar et al. have used logs in their CIT plugin [10]. They have logged the formal parameters linked to the functions and global variables and stored them in a database. This information was then later used for subsequent phases of their solution. Their focus is limited to logging the information related to functions and global variables [10]. Vankeirsbilck et al. have developed an approach for regression testing of a GCC toolchain for embedded CPUs. Their primary goal was to use a GCC plugin to apply software-implemented control flow error detection techniques automatically. Their approach was verified by logging the information related to the execution status of their selected case studies. They have saved the logged information into a CSV file. It provides a structured overview of the executed case studies. However, they have only focused on logging the status of different case studies. The internal behaviour of a GCC plugin has not been considered [12].

The current studies only focus on approaches to log specific components related to their research. Nevertheless, they provide insight into the significance of logs. Moreover, it is evident that logs can be utilised in software testing, debugging, verification, troubleshooting, etc.

## III. Discussion

We are currently working towards developing a testing framework to verify GCC plugins. It focuses on testing the GCC plugins on four different levels. It includes a static testing level, two dynamic testing levels focusing on white box and black box testing and one level focusing on keep-alive testing. The framework is code-named ”GCC-PT4” which stands for ”GCC - Plugin testing with 4 steps”. Working towards the white-box testing level made us realise that it is challenging to use traditional software testing frameworks such as DejaGNU and Google Test to verify GCC plugins. Such frameworks are aimed towards testing conventional C and C++ applications at run-time [13] [14] yet, GCC plugins execute at compiler run-time. However, the source of the GCC plugin is written using C or C++. Therefore, it can be argued that the source of the GCC plugin could be compiled as a conventional standalone C or C++ executable instead of a shared library. Such an executable could then be tested at run-time. Even though it is possible in theory, we learned that it is not practical. Most GCC plugins interact with GCC internals such as GIMPLE, RTL, etc. These GCC internal structures are generated during the compilation process. Therefore, such data structures which are generated during the compilation process are practically impossible to create or mock at application run-time. The work of De Blaere et al. [15] and the work of Vankeirsbilck et al. [2] provide clear examples of how plugins work with the GCC internal structures that cannot be mocked or created at application run-time.

To achieve white-box testing, we were motivated to explore out-of-the-box options. One considered option was the white box testing based on logs. Further into our research, it became apparent that logging is the gateway to observing the internal behaviour of GCC plugins. Since GCC plugins are written using C or C++, in principle, it should be possible to use existing logging frameworks for C or C++ such as spdlog, glog, Log4cxx, etc. We delved into existing logging frameworks. We explored the possibility of using such frameworks with GCC Plugins. However, we discovered that, in practice, existing frameworks cannot be used with GCC plugins due to multiple factors.

### A. Address space

GCC plugins share the same address space in which the GCC process operates. Therefore, the plugin has access to all global and static variables in this space. Problems such as data races and inaccurate log messages could occur when the logging framework is not designed to share the same address space [16].

### B. Exceptions & Run-Time Type Information (RTTI)

Logging frameworks rely mostly on complex features such as exceptions and RTTI. GCC plugins are often compiled with ”-fno-exception” and ”-fno-rtti” flags to ensure that they are small in size and will not impact the performance of the GCC compiler. These flags disable the exceptions and RTTI. Furthermore, frameworks use the serialisation and deserialisation of data structures via JSON or XML libraries. This would not work due to the unavailability of RTTI. Therefore, it is impossible to use frameworks relying on exceptions and RTTI [17].

### C. Multi-threaded environments

Generally, GCC plugins are used in multi-threaded environments. Existing logging frameworks often are not designed to be thread-safe. Moreover, they are not capable of handling the high level of concurrency found in multi-threaded environments. Hence, it is not possible to use frameworks that are not thread-safe [10].

### D. Library dependencies and system calls

Existing frameworks use libraries within. Most of those libraries use system functions. The most notable example is the use of ”malloc”, ”calloc” and ”free”. These library calls are not accessible to the GCC plugin API. Therefore, the compiler could crash, could cause unwanted delays, etc. [18].

### E. Environment of execution

Current frameworks are designed for traditional C and C++ applications. Conventional programs execute at run-time. However, GCC plugins execute at compile time. Conventional applications execute in a different environment to that of GCC plugins. Moreover, GCC plugins have access to different resources from that of conventional applications. Therefore, conventional frameworks would not be compatible with GCC plugins [3].

Out of many other factors, we believe above mentioned factors to be the most significant. They outline the drawbacks of existing frameworks towards logging in GCC plugins. However, logging remains crucial for GCC plugins.

Notably, it is the instrument that allows the programmers to monitor the internal execution of GCC plugins. Therefore, GCC plugin-focused logging is a necessity. Hence, it should be explored more. Additionally, it opens the door to a GCC plugin-focused logging framework.

## IV. Logging for GCC Plugins

We are developing a logging module as a part of ”GCC-PT4”. The focus of this module is to log necessary information during the execution of the GCC plugin. Then we intend to use the extracted information to achieve white-box testing. The logging module of ”GCC-PT4” is being designed and developed, focusing only on the GCC plugins. Conventional C or C++ programs have access to a comprehensive set of high-level programming features. This is not the case with GCC plugins. GCC plugins are developed with the compiler interaction as its main focus. Therefore, though GCC plugins are programmed with C or C++, GCC plugin API is limited to a specialised set of features. Thus, our logging module is specifically designed to use the components and libraries that are exclusively available to the GCC plugin API. This would help us overcome the bottlenecks of traditional frameworks. Our logging module focuses on logging the information related to functions, variables, errors and events within the GCC plugin. The logging module has a separate configuration. Current configuration settings allow the programmers to enable or disable logging, and control writing log information to a file and the console.

Currently, the plugin under test has two modes: test mode and standard mode. The modes are being controlled with the use of a custom compilation flag ”TEST_MODE”, used during the compilation of the plugin. The flag must be available and set to ”TEST_MODE=true” to activate logging in the GCC plugin. If the flag is unavailable or is set to ”TEST_MODE=false”, the plugin will be complied into the standard mode with logging disabled. The logging is enabled once the plugin is in test mode. Based on the configuration, the logging module will write the information to the console or the log file. Programmers can log information related to functions, variables, errors and events. As per their requirements, the programmers can decide which information they wish to log. Logged information could later be employed towards activities such as debugging, testing, etc. The utilisation of the custom compilation flag ”TEST_MODE” helps us ensure the plugin remains lightweight in its standard mode. Furthermore, this guarantees that both the plugin and compiler’s performance remain unaffected by the logging module. Presently, the only drawback in this solution is that the GCC plugin should be compiled twice, once for testing purposes and once for standard execution. Figure 2 provides a graphical overview of the above discussed.

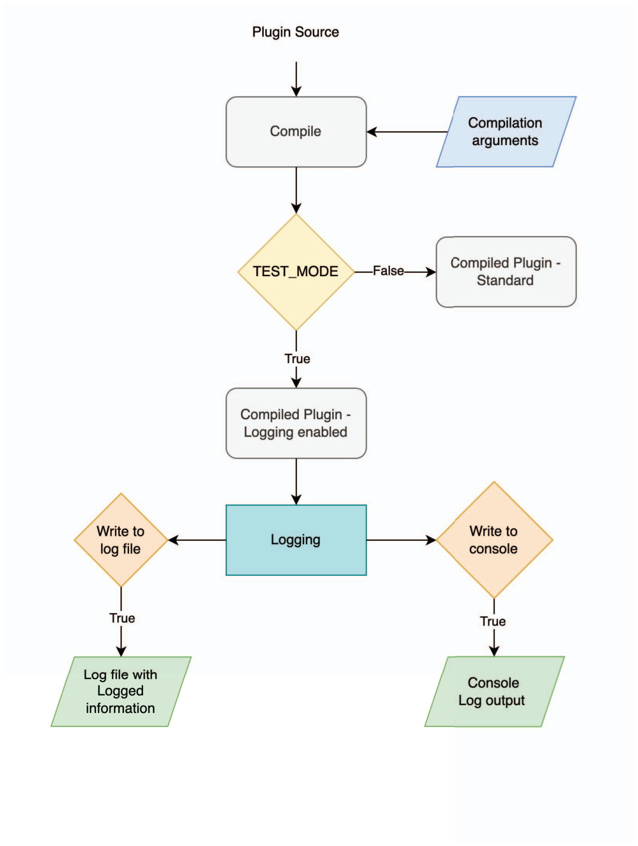

Fig. 2. Workflow of the logging module

## V. Conclusions & Future work

GCC is a powerful compiler. However, out-of-the-box features provided by the GCC were not sufficient. With GCC 4.5, GCC plugins became available as extensions to GCC. GCC plugins are written in C and C++. Therefore, one might assume that traditional logging frameworks for C and C++ could be used during plugin development. However, practically, this is not the case. This is due to multiple factors. We emphasised the importance of logging in GCC plugins. We illustrated how logging would enable the programmers to see the internal behaviour of a GCC Plugin.

We presented available literature on this domain. Moreover, we introduced our work in progress, ”GCC-PT4”. Furthermore, we highlighted the utilisation of logs towards achieving white box testing. We also presented the current structure of our logging module. Furthermore, we put forth how our module could overcome the challenges in the traditional logging frameworks.

Our ”GCC-PT4” and its logging module are still in the early stages of the development cycle. More effort is needed to develop our logging module into a more comprehensive logging framework. Additionally, it has to be tested with different plugins as well as with various environments. We intended to use existing GCC plugins such as that of De Blaere et al. [15] and Vankeirsbilck et al. [2] to verify our solution. Their plugins interact with the compilation process and modify GCC internals. Therefore, they offer to be ideal use cases to verify our solution. Furthermore, we aim to make additional contributions to the research community with our forthcoming work.

## Acknowledgment

This research is partially funded by the Research Fund KU Leuven.

## References

[1] W. Von Hagen, *The definitive guide to GCC*. Apress, 2011.
[2] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, “Random additive signature monitoring for control flow error detection,” *IEEE transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, 2017.
[3] (2018) plugins - gcc wiki. [Online]. Available: https://gcc.gnu.org/wiki/plugins
[4] C. Sun, V. Le, Q. Zhang, and Z. Su, “Toward understanding compiler bugs in gcc and llvm,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 294–305.
[5] (2023) Gcc 4.5 release series — changes, new features, and fixes - gnu project. [Online]. Available: https://gcc.gnu.org/gcc-4.5/changes.html
[6] (2010) Gnu compiler collection (gcc) internals. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.5.0/gccint/
[7] (2023) Plugins building (gnu compiler collection (gcc) internals). [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Plugins-building.html
[8] (2010) Plugin api (gnu compiler collection (gcc) internals). [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html
[9] G. Rong, Q. Zhang, X. Liu, and S. Gu, “A systematic review of logging practice in software engineering,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 534–539.
[10] S. S. Kumar, A. Chahar, and R. van Leuken, “Cit: A gcc plugin for the analysis and characterization of data dependencies in parallel programs,” in *Proceedings of the International Conference Design of Circuits and Integrated Systems*. Citeseer, 2013, pp. 1–6.
[11] S. Callanan, D. J. Dean, and E. Zadok, “Extending gcc with modular gimple optimizations,” in *Proceedings of the 2007 GCC Developers’ Summit*. Citeseer, 2007, pp. 31–37.
[12] J. Vankeirsbilck, J. Van Waes, H. Hallez, and J. Boydens, “Automated regression testing of a gcc toolchain used on embedded cpu programs,” in *2019 IEEE XXVIII International Scientific Conference Electronics (ET)*. IEEE, 2019, pp. 1–4.
[13] H. Cheddadi, S. Motahhir, and A. E. Ghzizal, “Google test/google mock to verify critical embedded software,” *arXiv preprint arXiv:2208.05317*, 2022.
[14] E. Albrecht, F. Gumz, and J. Grabowski, “Experiences in introducing blended learning in an introductory programming course,” in *Proceedings of the 3rd European Conference of Software Engineering Education*, 2018, pp. 93–101.
[15] B. De Blaere, E. Verstappe, J. Vankeirsbilck, and J. Boydens, “A compiler extension to protect embedded systems against data flow errors,” in *2021 XXX International Scientific Conference Electronics (ET)*. IEEE, 2021, pp. 1–6.
[16] Named address spaces. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/named-address-spaces.html
[17] I. Skochinsky, “Compiler internals: Exceptions and rtti,” *Also available as http://go. yurichev. com/17294*, 2012.
[18] Memory. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/memory.html