Packages Not Using The Default Build Flags: A Taxonomy

Emanuele Rocca
Arm
emanuele.rocca@arm.com

***Abstract*—While enabling the arm64 security feature known as PAC/BTI in Debian, we found that a sizable number of packages in the archive do not use the default build flags chosen by the distribution. The set of default flags is carefully selected by Debian to strike a balance between performance optimization, security, and ease of debugging. Packages setting their own flags may be affected by various issues in terms of performance, security, build reproducibility, and more. Additionally, the infrastructure available in Debian to set default build flags allows one to perform distribution-wide experiments, such as for example testing the impact of a new optimization flag. In this paper, we describe the issue of packages not using the default build flags, and categorize these deviations by their underlying causes.**

***Index Terms*—Build Flags, Linux Distributions, Debian, GCC**

I. Introduction

Computer programs written in compiled languages such as C, C++, and Fortran need to be compiled and linked to produce an executable file that the machine can run. The behavior of compilers and linkers can be controlled using command-line arguments known as "build flags", or "compiler flags". Several broad categories of build flags can be identified, including: performance optimization, security, adherence to standards, error handling, and more. Software developers use a set of build flags suitable for debugging purposes while writing and testing their code, and a different set of flags aimed at distributing the project once they are satisfied with the quality of their work and want to produce a release.

On Unix-like systems, it is common practice to use certain well-defined environment variables to specify the desired build flags depending on the programming language. For example, CFLAGS is the variable used to pass build flags to programs written in the C programming language, and CXXFLAGS is the variable to set for C++. The GNU Make manual describes the conventions for writing Makefiles for GNU programs, including the list of environment variables used to pass build flags, specify which compiler to use, and more [1]. Tools like GNU Automake, CMake, and Meson support such conventions. Linux distributions choose a set of build flags deemed appropriate for all packages they ship, and pass them to the build system via the environment variables described above.

We found a significant number of packages in Debian that do not follow the convention of using well-defined environment variables to select which build flags to use. Not using the build flags chosen by the distribution may have negative implications in terms of security, performance, and build reproducibility. This paper provides an analysis of the most common reasons packages do not use default build flags.

II. Methodology

A. Problem Statement

General purpose Linux distributions such as Debian, Fedora, or openSUSE provide a set of pre-built binary packages that users can install on their systems. Other distributions such as Gentoo are source based, meaning that the developers provide source packages for users to build and install. In both cases, distribution developers decide on a set of default build flags to use when building packages.

On Debian systems, the program `dpkg-buildflags` is used to retrieve the default compilation and linking flags. The concept of building packages with *hardened* security flags was first introduced while working on the Lenny release [2]. During the Wheezy release cycle, `dpkg-buildflags` was extended to return hardening build flags by default [3]. Table I shows the full list of environment variables set by `dpkg 1.22.20`, released in June 2025.

Environment Variable	Sets options for
`CFLAGS`	C compiler
`CPPFLAGS`	C preprocessor
`CXXFLAGS`	C++ compiler
`DFLAGS`	D compiler
`FFLAGS`	Fortran 77 compiler
`FCFLAGS`	Fortran 9x compiler
`OBJCFLAGS`	Objective C compiler
`OBCXXFLAGS`	Objective C++ compiler
`ASFLAGS`	Options for the assembler
`LDFLAGS`	Options for the compiler when linking

TABLE I
Build Flags set by Debian

The default flags can be overridden in several ways. For example, users can tweak the system-wide configuration file `/etc/dpkg/buildflags.conf` to set their own flags. This allows one to simply perform distribution-wide experiments, such as rebuilding all packages with custom flags.

We will now take a look at some of the options defined for programs written in the C programming language to illustrate the importance of honoring the default flags in terms of ease of debugging, performance optimization, build reproducibility, security, and code quality.

The default set of `CFLAGS` includes `-g` to add debugging information to the compiled binaries; this information is then typically included in a separate debug package with suffix `-dbgsym` and removed from the ELF files shipped by the main binary [4]. The performance optimization level chosen by default for all packages is `-O2`, generally believed to be a good compromise between performance, stability, and code size. The `-ffile-prefix-map` argument ensures that the build path is excluded from all generated files, which is an important step for making the build reproducible [5]. Using a reproducible build path also makes it possible to implement source code indexing for the purposes of automatically downloading debugging symbols from debuginfod servers [6]. The flags `-Wformat -Werror=format-security` are security-related settings included to ensure that all warnings about incorrect format strings such as those used by `printf` and `scanf` are considered build errors. To further reduce attack surface, `-fstack-protector-strong` and `-fstack-clash-protection` are used to mitigate stack smashing and stack clash exploits, respectively.

The set of flags is architecture-dependent; some build options are only available on certain architectures and not on others. The default flags on arm64 include two architecture-specific security features called PAC/BTI aimed at protecting Control-flow integrity [7], enabled with the flag `-mbranch-protection=standard`. Binaries built with PAC/BTI include a special ELF note indicating that the features are enabled. We perform daily checks on the Debian archive to track the progress of PAC/BTI enablement. Our analysis of the data revealed that, despite the inclusion of `-mbranch-protection=standard` in the set of default flags, a significant number of Debian packages are not built with the expected features enabled. The observation served as the primary motivation for this paper.

B. Proposed Approach

Given the key insight that binaries built with `-mbranch-protection=standard` include a special ELF note, we can analyze all ELF files in the Debian archive by checking their notes section for PAC/BTI enablement. As shown in Figure 1, only 50% of the 71000 binary packages in the archive are architecture-dependent. Moreover, not all architecture-dependent packages ship ELF files: out of 36000, about 12000 do not. Our analysis can thus focus on approximately 24000 packages in the Debian archive, a third of the total.

The methodology we follow to analyze the Debian archive for PAC/BTI enablement is:

1) Download all architecture-dependent binary packages from the Debian archive
2) For each package, check if it ships at least one ELF file
3) If it does, check if at least one ELF file has the note advertising PAC/BTI support
4) If no ELF files have PAC/BTI, the package requires further inspection

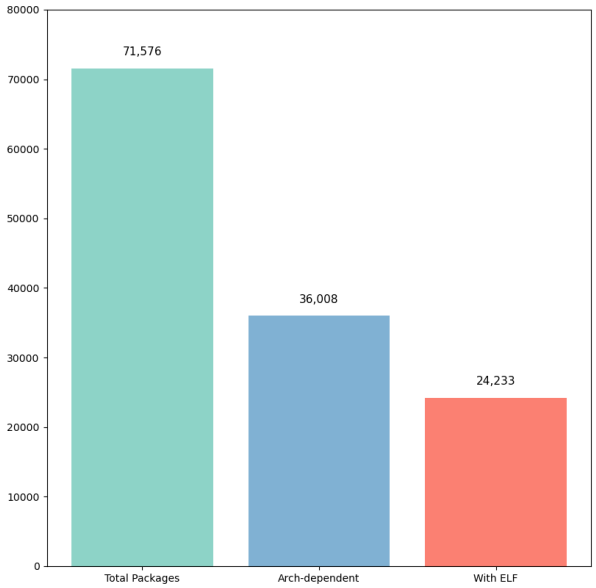

Fig. 1. Binary packages shipping ELF files

The majority of Debian packages shipping ELF files have the features turned on. Some do not, and there are various reasons to explain why that is the case. Some compilers producing ELF files simply do not support the features at this point. The Haskell, Go, OCaml, Rust, and Pascal compilers fall into this category.

There are however about 600 source packages built by compilers that do support PAC/BTI, and yet do not ship ELF files with the features on.

Debian packages are built by a network of build servers (*buildd*) [8], and build logs for each supported architecture are publicly available. We can thus fetch the latest arm64 build logs for all the 600 packages under examination and see if they contain `-mbranch-protection=standard`. Considering that the flag is included among the default ones set by Debian, all packages built without it likely ignore the default settings and set their own, either accidentally or on purpose.

III. Results

By inspecting the source code of packages built without the default build flags, we found the main causes to be the following:

A. Hand-written Makefiles
B. Misconfigured build systems
C. Ancient debhelper usage
D. Flags hardcoded in `debian/rules`

Issues A and B originate in the upstream codebase and therefore affect all downstream distributions that package the software, not only Debian. Conversely, issues 3 and 4 are specific to the Debian packaging and thus affect only Debian and possibly its derivatives.

A. Hand-written Makefiles

Consider the following excerpt from a real hand-written Makefile found in Debian:

```
1 CC = gcc
2 CFLAGS = -Wall -Wstrict-prototypes -O2
3
4 .c.o:
5   $(CC) $(CFLAGS) -c $<
```

Although on line 5 the Makefile does use the `CC` and `CFLAGS` variables, they are set unconditionally on line 1 and 2 respectively, overriding the values set in the environment (if any). A better approach consists in using make's *Conditional Variable Assignment* operator (`?=`) to define the variables only if they are not already set. The *Append* operator (`+=`) can also be used where appropriate. [1]

In the case of relatively simple software, there is a tendency to manually create a correspondingly simple Makefile. Most examples found online look similar to the snippet shown above, reinforcing the idea that unconditionally overriding build-related environment variables seems to be common practice when writing Makefiles by hand.

Notably, the example shown here also overrides the choice of compiler. Due to the widespread use of this approach, the project attempting a Debian-wide rebuild using Clang renames the Clang executables to match the names of their GCC counterparts, rather than relying on the `CC` and `CXX` environment variables [9].

B. Misconfigured build system

Most popular build systems, such as GNU Autotools and CMake, honor the various build-related environment variables by default. However, our analysis found some recurring misconfigurations. Taking CMake as the first example, let us consider the following, minimal `CMakeLists.txt`:

```
1 project(SimpleExample C)
2 add_executable(main main.c)
```

Using this example and passing the `CFLAGS` variable to CMake, we can confirm that indeed the chosen value gets used by the compiler when running `make VERBOSE=1`.

CMake uses the `CFLAGS` environment variable value in combination with its own builtin default flags for the toolchain to initialize and store the `CMAKE_C_FLAGS` cache entry. We have found various programs setting their own build flags as follows:

```
1 set(CMAKE_C_FLAGS "-Wall -O3 -pthread")
```

In this case the CMake configuration does not override the `CFLAGS` environment variable directly, as previously shown in section III-A, *Hand-written Makefiles* – the effect is however exactly the same: the build flags set by the user (or by the distribution) are ignored, and those specified by the upstream developer are used instead. The solution to this issue is analogous to the Makefile example, namely appending to the variable instead of overriding it:

```
1 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS}  -Wall
    -O3 -pthread")
```

Similarly, we have found that some projects using the GNU Autotools manually set the value of `CFLAGS` in `configure.ac`, overriding any previous value the variable may have had. In this case, once again, it is better to append the desired defaults to the variable instead.

We have not found any misconfigurations in packages using the Meson Build system, which is likely attributable to its provision of explicit mechanisms for specifying default build flags, such as `add_project_arguments`.

C. Ancient debhelper usage

The vast majority of software in Debian is packaged using `debhelper`, a suite of tools automating various common aspects of building packages. Since the introduction of the debhelper command sequencer *dh(1)* in 2008, many `debian/rules` files have become extremely simple, in some cases just a couple of lines.

One of the features of the sequencer is automatically setting all environment variables related to build flags listed in table I, unless they are already set. A few packages in Debian do not adopt the dh short style format, using debhelper the old-fashioned way instead. By following such approach, the variables containing the default build flags set by `dpkg-buildflags` are not automatically set. Maintainers can either call `dpkg-buildflags` directly, or use the following snippet:

```
1 DPKG_EXPORT_BUILDFLAGS = 1
2 include /usr/share/dpkg/buildflags.mk
```

Furthermore, debhelper adopts the concept of *Compatibility Levels* in order to ensure that major backwards-incompatible changes do not break existing packages. Automatic setting of build flags was introduced in 2012 when debhelper bumped its compatibility level to 9. All packages declaring a compatibility level less than 9 also need to manually set the build flags as described for those not using the dh short style format. Debian Trends offers historical insights into the adoption of debhelper compatibility levels. [10]

D. Flags hardcoded in debian/rules

In certain cases, the upstream build system is configured correctly, but the way the Debian packaging tools are used leads to build flags that differ from Debian's default values. One such rather obvious example is when the maintainer explicitly sets the build flags manually in `debian/rules`. We have found several such cases in our analysis.

Another, more subtle circumstance occurs when the maintainer tries to append to a variable, for example `CFLAGS`, but does so before the automatic setting of the variable by `dh` takes

place. Consider the following, simplified snippet inspired by a real case:

```
1 #!/usr/bin/make -f
2
3 export CFLAGS += -pipe -fPIC -Wall
4
5 %:
6   dh $@
```

The intention of the maintainer is clear: they want to append some values to the default build flags, not override them. However, the variable is not set yet on line 3, debhelper only sets it later on. Crucially, it does so only if it is not already set. Given that the snippet sets it, the end result is that the package is built with `-pipe -fPIC -Wall`, and not the default flags. Precisely for this use case, `dpkg-buildflags` provides the `DEB_CFLAGS_MAINT_APPEND` variable, which should be used instead.

IV. Related Work

To the best of our knowledge, this is the first work to analyze the outcome of a complete distribution rebuild to investigate packages that deviate from the default build flags.

Previous work has instead focused on detecting the compiler optimization levels used when building a binary [11] [12], exploring optimal compiler flags for performance [13] [14], energy efficiency [15], and fault tolerance [16].

Various authors have focused on mass rebuilding several packages in a distribution, either to search for potential security flaws [17], to parallelize the work [18], or to evaluate the efficiency of the process itself [19].

There is an open proposal by the Fedora community to add markers to ELF objects with the goal of determining whether they have certain properties, including some of the build flags they have been built with. [20] Such markers would be a great help when trying to single out binaries built without the default flags.

V. Conclusion

This work started from the observation that a non-negligible number of packages in Debian is built without the default build flags set by the distribution. We used the knowledge that a special ELF note is added to binaries built with one of the default flags set by Debian on arm64. Checking for the absence of this note allowed us to reduce the scope of analysis from several thousand source packages producing ELF files to a few hundreds.

We then inspected the source code of packages built without the default build flags and found four main causes, two related to the upstream code (Hand-written Makefiles and misconfigured build systems) and two found in the Debian packaging (ancient debhelper usage, flags hardcoded in `debian/rules`).

We hope that this paper provides useful information to help software authors and package maintainers improve the quality of Debian packages.

References

[1] R. M. Stallman, R. McGrath, and P. Smith, "Gnu make," *Free Software Foundation, Boston*, 1988.

[2] M. Muehlenhoff, "Introducing security hardening features for lenny," 2008, accessed: 2025-06-16. [Online]. Available: https://lists.debian.org/debian-devel-announce/2008/01/msg00006.html

[3] R. Hertzog, "Bits from dpkg developers - dpkg 1.16.1," 2011, accessed: 2025-06-16. [Online]. Available: https://lists.debian.org/debian-devel-announce/2011/09/msg00001.html

[4] "Automatic debug packages," 2025, accessed: 2025-06-06. [Online]. Available: https://wiki.debian.org/AutomaticDebugPackages

[5] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.

[6] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[8] M. Caneill and S. Zacchiroli, "Debsources: Live and historical views on macro-level software evolution," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10.

[9] S. Ledru, "Rebuild of the debian archive with clang," 2025, accessed: 2025-06-12. [Online]. Available: https://clang.debian.net/

[10] L. Nussbaum, "Trends: Debhelper compatibility level," 2025, accessed: 2025-06-18. [Online]. Available: https://trends.debian.net/#debhelper-compatibility-level

[11] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, "Himalia: Recovering compiler optimization levels from binaries by deep learning," in *Intelligent Systems and Applications: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 1*. Springer, 2019, pp. 35–47.

[12] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163 461–163 475, 2021.

[13] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013.

[14] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 165–174.

[15] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2015.

[16] R. Thunig, M. Johannfunke, T. Wang, and H. Schirmeier, "One flag to rule them all? on the quest for compiler optimizations to improve fault tolerance," in *2024 19th European Dependable Computing Conference (EDCC)*. IEEE, 2024, pp. 33–40.

[17] J. Ruohonen, M. Saddiqa, and K. Sierszecki, "A static analysis of popular c packages in linux," *arXiv preprint arXiv:2409.18530*, 2024.

[18] L. Nussbaum, "Rebuilding debian using distributed computing," in *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, 2009, pp. 11–16.

[19] S. R. Tate and B. Yuan, "On the efficiency of building large collections of software: Modeling, algorithms, and experimental results," in *International Conference on Software Technologies*. Springer, 2022, pp. 145–168.

[20] "Fedora toolchain watermarks," 2025, accessed: 2025-06-13. [Online]. Available: https://fedoraproject.org/wiki/Toolchain/Watermark