# On Build Hermeticity in Bazel-based Build Systems

Shenyu Zheng, *Queen's University, Canada*

Bram Adams, PhD., *Queen's University, Canada*

Ahmed E. Hassan, PhD., *Queen's University, Canada*

*Abstract—A hermetic build system manages its own build dependencies, isolated from the host file system, thereby securing the build process. Although, in recent years, new artifact-based build technologies like Bazel offer build hermeticity as a core functionality, no empirical study has evaluated how effectively these new build technologies achieve build hermeticity. This paper studies 2,439 non-hermetic build dependency packages of 70 Bazel-using open-source projects by analyzing 150 million Linux system file calls collected in their build processes. We found that none of the studied projects has a completely hermetic build process, largely due to the use of non-hermetic top-level toolchains. 71.9% of these are Linux utility toolchains that in principle could be managed by Bazel, while 38.1% are programming language-related toolchains introduced by the default configuration of key Bazel build rules. Furthermore, we evaluate the risks of non-hermetic build dependencies when building projects on new machines or within CI.*

**Modern software systems** rely on a large number of dependencies (i.e., libraries and build tools) during compilation, many of which are open-source projects, making the build system an ideal target for supply chain attacks [1] [2] [4]. To secure the build process, the Supply-chain Levels for Software Artifacts (SLSA) framework[1], based on Google's internal processes, stipulates that an ephemeral, isolated and hermetic build environment is needed to achieve the highest level of supply chain security [1].

A hermetic build system, therefore, by downloading and managing all necessary toolchains and libraries at the build tool level, ensures a self-contained build environment that isolates (secures) the builds from changes to the dependencies (libraries and toolchains) installed on the host operating system. Hermeticity also lays the foundation for reproducible builds, i.e., builds that produce identical outputs when given the same set of inputs and build environment [3], and it is an important precondition for reliable parallel and incremental builds (in addition to other preconditions, like the need to carefully manage the build outputs of previous builds).

Despite its importance, implementing a hermetic build system is non-trivial. Traditional file-based (e.g., Make) and task-based (e.g., Maven) build technologies do not have built-in support for build isolation, meaning that changes external to the build system can influence the build results [5]. This lack of isolation also makes it challenging to enforce dependency management as access to undeclared dependencies cannot be tracked by the build system [13], leading to non-deterministic and inconsistent build results. Furthermore, while these build technologies are often capable of managing libraries, they often do not support managing toolchains, requiring developers to continuously maintain their build systems to adapt to new environments [11].

An alternative solution is for developers to use external tools such as continuous integration (CI), Infrastructure-as-Code (IaC), or functional package manager (e.g., Nix, Guix) tools to create a fully managed and consistent environment for the build process [13]. However, since these tools are external to build tools, the specification of the entire build environment typically is spread across multiple configuration

XXXX-XXX © 2024 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

[1] https://slsa.dev/spec/v1.0/

files (e.g., requirements.txt + Dockerfile + Jenkinsfile), which results in additional build comprehension and maintenance effort.

In recent years, a new family of artifact-based build system technologies (e.g., Bazel[2], Pants) has emerged, introducing an innovative approach for enhancing build performance and correctness, as well as facilitating build hermeticity. Contrary to traditional build systems like Make and Maven, artifact-based systems like Bazel require developers to define what artifacts and dependencies need to be built or installed without specifying how, leaving the build system to manage task configuration, scheduling, and execution. Consequently, the build system has full control over the build process and knowledge of build dependencies, ensuring deterministic builds and enabling build hermeticity [14].

Despite artifact-based build technologies' claim of hermetic builds, it remains uncertain how effectively they are able to achieve this goal. In fact, a recent study [8] shows how developers are abandoning modern build technologies like Bazel as their perceived benefits do not outweigh their maintenance costs. Recent studies trying to understand this phenomenon mostly studied Bazel's build performance [14] and maintenance overhead [8]. Moreover, while Bazel employs different strategies to ensure the hermeticity of the builds (e.g., performing builds in a sandbox), their effectiveness relies on the underlying build rules. These rules, especially when they are either customized by developers or obtained from the community, may not be hermetic. Therefore, understanding the hermeticity of Bazel builds in practice becomes crucial in aiding developers' decision-making regarding the adoption of Bazel.

This paper first performs a dynamic analysis of 150 million filesystem calls gathered during the Bazel build processes of 70 open-source projects to assess the hermeticity of these builds. For those projects exhibiting non-hermetic builds, we investigate whether the non-hermetic build dependencies are instead documented outside the Bazel specification files, i.e., Dockerfiles or CI configuration files. Subsequently, we explore the causes of non-hermeticity, offering insights for developers on improving build hermeticity. Lastly, we evaluate the potential impact of non-hermeticity in practice, by examining the risks of non-hermetic dependencies being installed on new machines or being unexpectedly updated by CI environments.

Despite Bazel's promises, our results show that none of the studied projects exhibited a completely hermetic build process. In addition, while 28.57% of projects employ external tools to manage build dependencies not handled by Bazel, their management approach still lacks hermeticity. We also found that non-hermetic build dependencies may not be pre-installed on new build environments and could experience unexpected updates within popular third-party CI environments. Bazel users aiming to improve build hermeticity should closely examine the default (often non-hermetic) configuration of their Bazel build rules. As Linux utility toolchains are often overlooked and not managed by Bazel, developers should use hermetic build rules such as *toolchain_utils* and *rules_tar* to manage them.

[2]https://bazel.build/

## Analyzing the Hermeticity of Bazel Builds

This section explains how we collected the data on Bazel build hermeticity for the three research questions, while question-specific analyses are discussed in the results section. In our previous study [14], we identified 70 non-trivial buildable Bazel projects. As illustrated in Fig 1, we use the `strace` command to record the Linux file system calls made in the build processes of these 70 projects. Then, we analyzed the file system calls to identify whether the accessed packages were managed by Bazel or by the underlying host. The experiments were performed on a Debian 11 server with 8 vCPUs and 32 GB memory. The replication package is available online[3][4].

### Collect Linux File System Calls

As large projects can take hours to build and produce extremely large `strace` logs, we limited the range of tracked system calls to the subset related to file operations. To ensure our selection was representative, we focus on the `strace` file operation system calls that are among the top 100 most frequently used system calls, as identified by Tsai et al. [10]. Since we want to identify the files created, read, modified or executed by the build process, we focused on the most frequently-used IO-related system calls (e.g., open, read), while we excluded file system calls that ignore file contents, e.g., stat and getcwd. To enhance the accuracy of our results, we also included variations of the selected system calls (e.g., openat).

[3]https://github.com/SAILResearch/replication-24-shenyu-bazel_hermeticity

[4]https://zenodo.org/doi/10.5281/zenodo.13324236

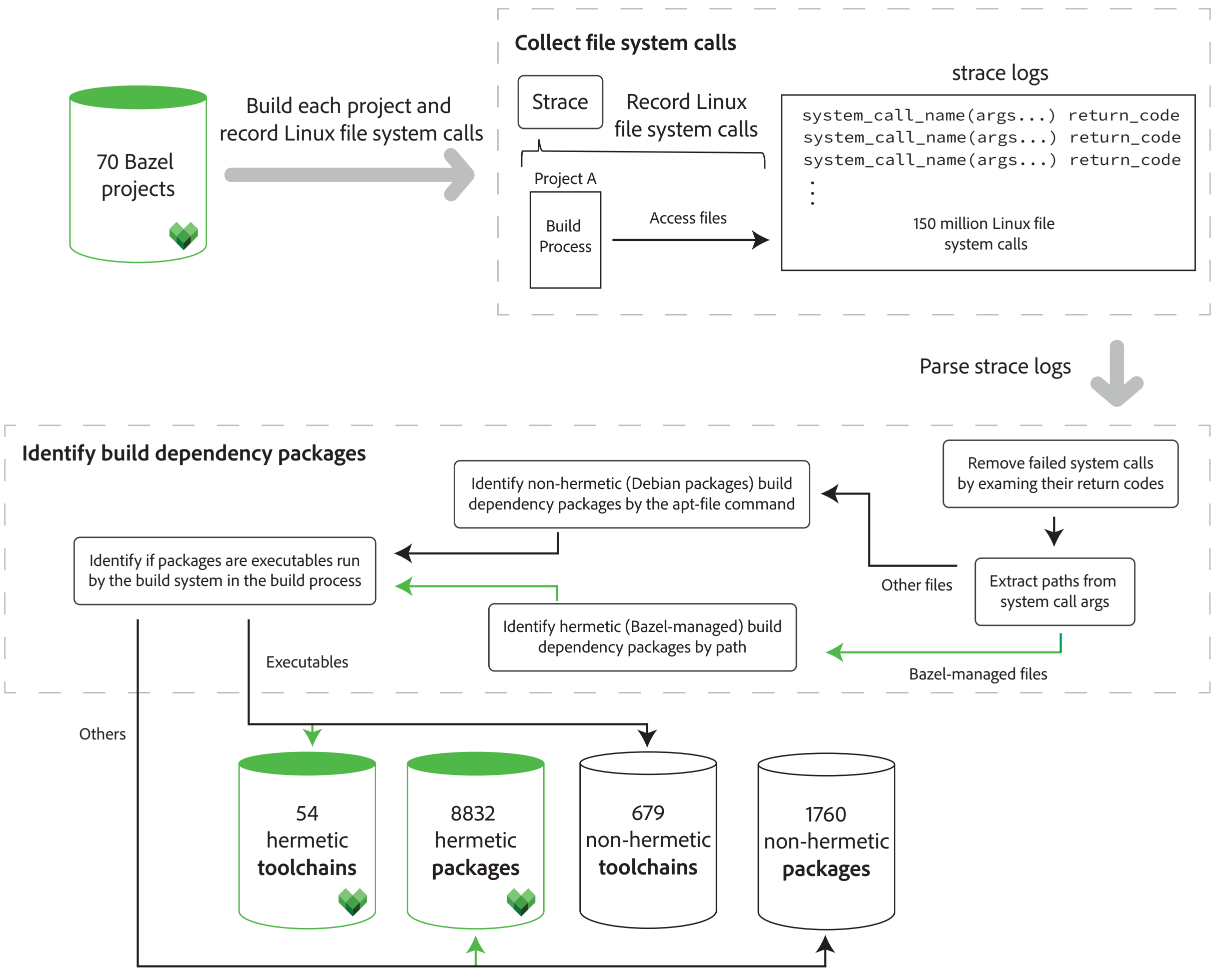
70 Bazel projects
Build each project and record Linux file system calls
Collect file system calls
Strace
Record Linux file system calls
Project A
Build Process
Access files
strace logs
system_call_name(args...) return_code
system_call_name(args...) return_code
system_call_name(args...) return_code
150 million Linux file system calls
Parse strace logs
Identify build dependency packages
Remove failed system calls by examing their return codes
Extract paths from system call args
Other files
Bazel-managed files
Identify non-hermetic (Debian packages) build dependency packages by the apt-file command
Identify hermetic (Bazel-managed) build dependency packages by path
Identify if packages are executables run by the build system in the build process
Executables
Others
54 hermetic toolchains
8832 hermetic packages
679 non-hermetic toolchains
1760 non-hermetic packages

**FIGURE 1.** The process of analyzing the hermeticity of Bazel builds.

The traced system calls listed below resulted in 150 million file system calls during the build process across the 70 projects.

```
strace ... -e trace=creat,open,openat,
rename,renameat, mkdir,mkdirat,rmdir,
link,linkat,symlink,symlinkat,
unlink,unlinkat,read,readv,write,
writev,execve,execveat ...
```

## Identify Build Dependency Packages

To evaluate the hermeticity level, we then identified the build dependencies (i.e., toolchains and libraries) accessed in the build process and classified them into **non-hermetic** (installed on the host system via operating system packages) and **hermetic** (installed in Bazel-managed directories by Bazel).

As shown in Figure 1, we first employed a script to extract the names, return codes, and the associated files of file system calls from the strace logs. In the build process, Bazel searches through multiple paths to locate packages, leading to Linux file system calls for non-existent files. To filter out such system calls, we only include file system calls that were successful, as identified by a return code other than -1. Then, according to the extracted paths, we categorized their associated files into two groups as described below.

- Hermetic (Bazel-managed) files - Files situated in the directory `$HOME/.cache/bazel` or projects' own directories are deemed hermetic since Bazel manages these files in the build process.
- Non-hermetic (Unmanaged) files - Other files that are not managed by Bazel.

In total, we identified 11,602,678 hermetic and 46,775 non-hermetic files accessed in the build process of the 70 Bazel projects.

As non-hermetic dependencies are usually installed via operating system packages, we mapped non-hermetic files to operating system packages, specifically Debian Linux packages, due to the widespread use of Debian-based Linux distribution in CI platforms (for example, the only available Linux distribution for the GitHub Actions runner is the Debian-based Ubuntu distribution). We executed the `apt-file search` command to identify the specific non-hermetic Debian Linux packages linked to 30,844 non-hermetic files.

Hermetic dependencies, which are installed by Bazel, are stored in the directory `$HOME/.cache/bazel/.../external/<package name>`[5]. Therefore, we retained 9,931,674 hermetic files located in such directories and identified their corresponding package names, while the remaining 1,671,004 files, which were mostly source code files or Bazel-created temporary files, were excluded from the analysis.

Once we had separated hermetic and non-hermetic build dependency packages, we proceeded to classify them based on their associated file system calls. Packages that have any file associated with a file system call that is *execve* or *execveat* were categorized as toolchains, whereas all others were identified as libraries.

Notably, some non-hermetic dependencies are only installed because they are transitively required by another dependency. Therefore, for each project, we employed the `apt-cache depends` command to query the transitive dependencies of non-hermetic packages and construct a dependency graph for these packages. The packages at the root node of the graph are top-level dependencies, while others are transitive dependencies.

### Analyze If the Non-Hermetic Build Dependencies Are Externally Managed

Considering that developers might employ external tools to handle their non-hermetic dependencies, we further analyzed whether these dependencies are managed by Docker and CI tools, given their widespread use in managing the build environment. To accurately identify the non-hermetic dependencies specified in Dockerfiles and CI configuration files, we employed a script to search all files containing the names of the non-hermetic packages. We then manually inspected the identified files to determine whether they were Dockerfiles or CI configuration files and to check if the non-hermetic dependencies were installed within these files. Additionally, for each identified non-hermetic dependency, we manually checked whether developers specified the version number in the configuration files when installing them.

[5] https://bazel.build/remote/output-directories

## RQ1. How Hermetic are Bazel Builds?

**Achieving a completely hermetic build is hard, even with artifact-based build system technologies like Bazel.** In our case study, we examined 70 Bazel projects and identified a total of 11,325 build dependency packages used during the build process. Among these, 8,886 were hermetic packages (8,832 libraries and 54 toolchains), and 2,439 were non-hermetic packages (1,760 libraries and 679 toolchains). Fig 2 (a) shows the percentage of projects that exhibit non-hermetic/hermetic Bazel build processes, as well as the number of packages, categorized as toolchains or libraries, that are managed (hermetic) and not managed (non-hermetic) by Bazel. The median numbers of non-hermetic libraries and toolchains across the projects are 21 and 9, respectively, compared to 51 and 0 for hermetic ones.

While Bazel and related build technologies like Pants claim build hermeticity as one of their main fortes, none of the projects analyzed in the dataset demonstrate a fully hermetic build process. Notably, the total number of non-hermetic toolchains is significantly higher compared to hermetic ones (Mann-Whitney U test at $\alpha$ = 0.01), with a large Cliff's Delta effect size (0.984). In the case of libraries, although the total number of non-hermetic libraries is significantly lower than hermetic libraries, its median still reaches a notable 21 non-hermetic libraries used in the build process.

**The non-hermetic top-level toolchains are the major source of non-hermeticity.** Among the 2,439 non-hermetic dependency packages, around 81.6% of them are transitive dependencies. Additionally, as shown in Fig 2 (a), the number of non-hermetic top-level toolchains is significantly higher than that of non-hermetic top-level libraries, with medians of 5 and 1, respectively. In fact, 54.3% of projects depend on non-hermetic top-level libraries in the build process, compared to 98.6% using non-hermetic top-level toolchains (numbers not shown in the figure). This suggests that the usage of non-hermetic top-level toolchains might contribute most to the access of host system resources in the build process.

**Bazel users rarely employ external tools to manage the non-hermetic top-level build depen-**

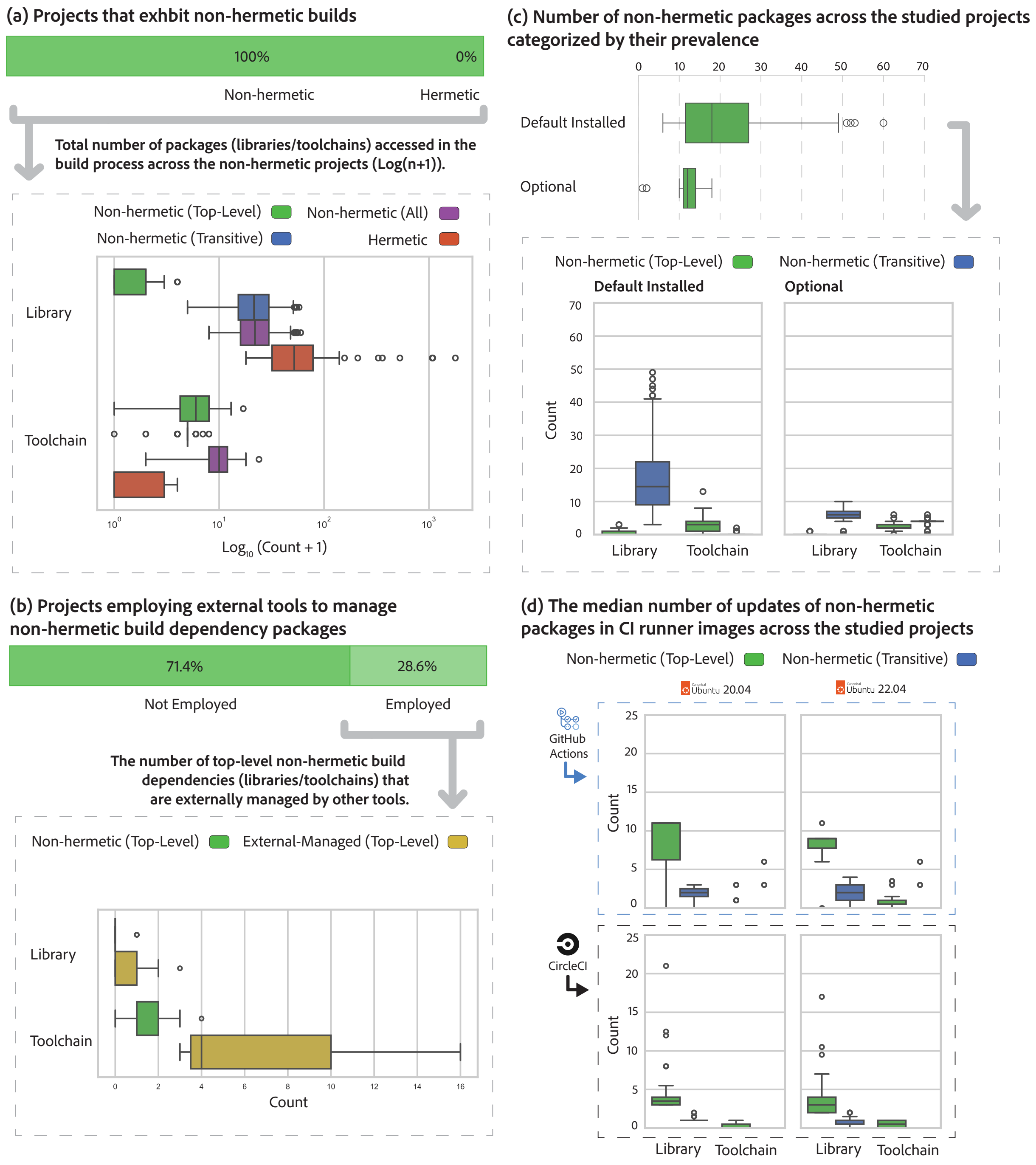

**FIGURE 2.** (a) The percentage of hermetic/non-hermetic projects, and the total number of libraries/toolchains accessed in the build process across the non-hermetic projects (Log(n+1)). (b) The percentage of projects that external tools manage their non-hermetic build dependencies, and the total number of non-hermetic top-level build dependencies that are externally managed by other tools. (c) The number of non-hermetic packages categorized by their Debian Priority - Default Installed (Debian Priority: *Required*, *Important*, and *Standard*), Optional (Debian Priority: *Optional*) across the 70 projects. (d) The distribution of a median number of updates of build dependencies within the Ubuntu 20.04 and 22.04 runner images for GitHub Actions and CircleCI.

**dencies; when they do, developers often omit dependency version constraints.** Fig 2 (b) illustrates the percentage of non-hermetic projects employing external tools (e.g., Docker, CI services) to manage the non-hermetic top-level build dependencies. While 28.6% of projects in the dataset also employ other tools to control their non-hermetic top-level libraries and toolchains used in the build, their number of non-hermetic libraries and toolchains is still significantly higher than the number of externally managed ones. This suggests that these projects are losing track of a substantial number of non-hermetic top-level libraries and toolchains.

Furthermore, among those 28.6% of projects that use external tools to control non-hermetic build dependencies, 70% of them do not specify the versions of those dependencies in their Dockerfiles or CI configuration files. This can also lead to non-hermeticity, as these projects will resort to using the latest available version of these toolchains and libraries in the build process, which could vary from build to build. For example, while *chipsalliance/verible* installs their build toolchain dependency `g++` by running `apt install build-essential` in their CI builds, as there is no version control, the version of `g++` used in the build process changes across builds.

**For developers:** Even with tools like Bazel, build hermeticity does not come for free. As non-hermetic top-level toolchains are a major source of non-hermeticity, only exacerbated by transitive dependencies, techniques like Software Bill of Materials (SBoM) can provide visibility into the full dependency tree, helping migrate non-hermetic transitive dependencies. Furthermore, when utilizing external dependency management tools like Docker, it is essential to specify the versions of these dependencies.

## RQ2. What are the reasons for non-hermetic build dependencies?

**Only 37.1% of projects are configured to use hermetic toolchains in their build configuration files.** To further understand the reasons behind non-hermetic dependency packages, particularly non-hermetic top-level toolchains, we manually analyzed each project's *WORKSPACE.bazel* and *MODULE.bazel* files, which define the external dependencies for Bazel. We examined whether the top-level toolchains of the project were specified, and analyzed their configuration by searching for the names of the toolchains. We found that only 37.1% of projects are configured to use hermetic toolchains, and that none of these manage all their toolchains with Bazel.

Additionally, we observed that all hermetic toolchains are programming language-related (e.g., JDK, Python), whereas more general Linux utility toolchains (e.g., Tar, Grep) are often overlooked, with 71.9% of the 394 non-hermetic top-level toolchains representing utilities. Although these utility toolchains are common in Linux installations, recent attacks targeting these tools (e.g., the XZ Utils backdoor[6]) demonstrate that leaving them unmanaged can expose projects to significant security threats.

**Non-hermetic programming language-related toolchains are introduced by the default configuration of official Bazel build rules** Although Bazel provides explicit support to manage toolchains for major programming languages, 38.1% of the identified non-hermetic top-level toolchains are programming language-related toolchains. One major reason for build processes' direct access to host system resources is the non-hermetic default configuration of certain official build rules.

For instance, the default configurations of the official *rules_cc* and *rules_python* build rules use the host-installed toolchains. While the *rules_java* build rule by default compiles a code base using a Bazel-managed JDK, it executes and tests said code base using a locally installed JDK, therefore the resulting binaries also depend on what is installed on the machine.

Although these default configurations are non-hermetic, few developers override them. Among the 13 projects using the *rules_python* build rule, only 1 project changed the default configuration to use Bazel to download and manage Python in the build process. The impact of the default configuration can be more obvious when we compare the *rules_python* build rule to the *rules_go* build rule, which, by default, uses a hermetic Go SDK. Out of the 24 projects that use the *rules_go* build rule, only 1 project changed the configuration to use a non-hermetic host-installed Go SDK.

**For the 37.1% of projects that use hermetic toolchains in their build process, 15.4% (4/26) of them have their hermetic toolchains accessing non-hermetic files, still leading to non-hermeticity.** Even if developers use Bazel to manage their toolchains, it may still introduce non-hermeticity. For example, in the case of *jesec/rtorrent*, CMake is executed using Bazel during the build process, but the underlying Make invoked by CMake is host-installed. Another factor contributing to the non-hermeticity of "hermetic" toolchains is cache files associated

[6]https://en.wikipedia.org/wiki/XZ_Utils_backdoor

with toolchains. We observed 3 instances such as *bazelbuild/starlark* accessing Cargo's caches, *bazelbuild/rules_jvm_external* accessing yarn's caches, and *lowRISC/opentitan* accessing pip's caches in the build process, with all of the accessed cache files being stored on the host machines.

**For developers:** Despite Bazel's claim of providing hermetic builds, achieving this requires substantial effort from developers. Since the default configurations of build rules such as *rules_cc* and *rules_python* are non-hermetic, developers must manually override them to use Bazel-managed toolchains. Additionally, while Linux utility toolchains are often overlooked, developers should use hermetic build rules such as *toolchain_utils* and *rules_tar* to manage them within build configuration files.

## RQ3. What is the impact of build hermeticity?

**Projects contain a median of 12 non-hermetic dependencies that are not by default included in the standard Debian installation, leading to potentially inconsistent build results or even failures when the projects are built on new machines.** In Debian, each package has a priority value, which is used to control if it is included in standard Debian installations. Packages with the *Required*, *Important*, or *Standard* priority are installed by default in the standard installation, while those with the *Optional* priority are only installed at the specific request of users. To measure the impact of non-hermetic dependencies in Bazel on rebuildability when building on different machines, we employed a script to classify these dependencies as default or non-default installations according to their Debian priority.

As illustrated in Fig 2 (c), although the number of default-installed build dependencies is significantly higher than the number of optional ones, the latter number still reaches a median of 12 across the 70 studied projects, whereas the median number for default-installed ones is 18. Additionally, non-hermetic top-level toolchains, identified in RQ1 as a major source of non-hermeticity, show a median count of 2 for optional non-hermetic top-level toolchains.

**For developers:** Developers risk build failures and higher maintenance efforts when working on systems lacking optional non-hermetic build dependencies. Furthermore, since optional dependencies require explicit installation and updates by developers, as there is often a lag between a package being released to repositories and being updated in Linux distributions or Docker images [9], there is a higher likelihood of inconsistencies in the versions of build dependencies installed across systems. This issue becomes more pronounced when using Bazel across various platforms.

**Bazel users may face unexpected changes in their non-hermetic build dependencies within CI environments.** The median numbers of updates for top-level libraries are 11 and 9 for the Ubuntu 20.04 and 22.04 GitHub Actions runner images, respectively, and 3.5 and 3.0 for CircleCI. In CI settings, non-hermetic dependencies that are neither managed by Bazel nor externally, have to be supplied by pre-installed packages on CI platforms, enabling the build to proceed successfully. While such builds appear to be hermetic, the pre-installed packages in the CI environment fall outside the control of developers. Changes to these pre-installed packages by the CI vendor can result in unexpected build failures.

To understand the potential impact of such an issue, we conducted further analysis on the two most popular CI services on GitHub, i.e., GitHub Actions and CircleCI [7]. Specifically, we examined the frequency of updates for the non-hermetic dependencies of the 70 projects installed in the Ubuntu 20.04 and 22.04 runners of the two CI services. For GitHub Actions, the packages installed on runners are specified in their README files[7]. We employed a script to analyze the commit history of README files for Ubuntu 20.04 and 22.04 runners to identify the version changes of installed packages over the last three years. In the case of CircleCI, we retrieved all the tags of their base Docker images for Ubuntu 20.04 and 22.04 runners, executing the `apt list -installed` command with the container created by each image to extract the history of the installed package versions. Fig 2 (d) illustrates the distribution of the median number of updates of build dependencies across both Ubuntu 20.04 and 22.04 runner images on GitHub Actions and CircleCI.

Despite non-hermetic top-level toolchains being identified as a primary source of non-hermeticity, the median update counts for these toolchains in the Ubuntu GitHub and CircleCI runner images for versions 20.04 and 22.04 are relatively low (0, 1, 0 and 0.5, respectively), although the maximum number of updates can reach up to 17. On the other hand, the update counts for non-hermetic libraries were higher, with median counts of 11 and 9 for top-level libraries in the 20.04 and 22.04 Ubuntu GitHub runner images,

[7]https://github.com/actions/runner-images/blob/main/images/ubuntu/Ubuntu2004-Readme.md

and 3.5 and 3.0 for the CircleCI images, respectively, while 2, 2, 1, and 1 for transitive libraries in the same respective runner images.

**For developers:** While Bazel's incremental build feature is a major selling point [8], promising significant build time reduction [14] [15], unexpected/uncontrolled changes to non-hermetic build dependencies could lead to incorrect build results and performance downgrade for incremental builds due to build cache invalidation. Although the update counts for non-hermetic build dependencies in CI runner images may seem low, when build failures or performance downgrades occur, diagnosing and addressing these issues can be complex, as such updates often go unnoticed due to the CI environment not being under their control and the inherent complexity of debugging CI builds [12].

## LIMITATIONS

As our findings are specific to Bazel-based artifact-based build systems, we plan to extend our research to both other artifact-based and task-based build systems on more projects. Additionally, files accessed by system calls were mapped to Debian packages and Bazel-managed packages to evaluate the hermeticity of Bazel. As Bazel might employ different approaches to achieve build hermeticity outside of Linux, such as Windows or MacOS, we also plan to conduct a broader evaluation on other platforms.

## CONCLUSION

Our research indicates that complete build hermeticity was not achieved in any of the Bazel projects we examined, largely due to the default configuration of the build rules. Furthermore, we found that the non-hermetic dependencies may not be included by default in the standard Debian installation, risking build failures on new build machines. Developers might also encounter unanticipated updates to their non-hermetic dependencies within CI environments, potentially leading to inconsistencies in builds or a decline in performance, as these dependencies can be inadvertently changed by the CI vendors. For developers, although artifact-based build technologies like Bazel offer rich features to enhance build hermeticity, it is important to go beyond their default configuration.

## REFERENCES

1. Enck, W. & Williams, L. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*. **20**, 96-100 (2022)
2. Fourné, M., Wermke, D., Enck, W., Fahl, S. & Acar, Y. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. *2023 IEEE Symposium On Security And Privacy (SP)*. pp. 1527-1544 (2023)
3. Randrianaina, G., Eddine Khelladi, D., Zendra, O. & Acher, M. Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems. *2024 IEEE/ACM 21st International Conference On Mining Software Repositories (MSR)*. pp. 654-664 (2024)
4. Butler, S., Gamalielsson, J., Lundell, B., Brax, C., Mattsson, A., Gustavsson, T., Feist, J., Kvarnström, B. & Lönroth, E. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal*. **31**, 687-719 (2023)
5. Lamb, C. & Zacchiroli, S. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*. **39**, 62-70 (2021)
6. Bajaj, R., Fernandes, E., Adams, B. & Hassan, A. Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors. *Empirical Software Engineering*. **29**, 11 (2023,11)
7. Golzadeh, M., Decan, A. & Mens, T. On the rise and fall of CI services in GitHub. *2022 IEEE International Conference On Software Analysis, Evolution And Reengineering (SANER)*. pp. 662-672 (2022)
8. Alfadel, M. & McIntosh, S. The Classics Never Go Out of Style: An Empirical Study of Downgrades from the Bazel Build Technology. *2024 IEEE/ACM 46th International Conference On Software Engineering (ICSE)*. pp. 1017-1017 (2024)
9. Zerouali, A., Mens, T., Decan, A., Gonzalez-Barahona, J. & Robles, G. A multi-dimensional analysis of technical lag in Debian-based Docker images. *Empirical Software Engineering*. **26**, 19 (2021)
10. Tsai, C., Jain, B., Abdul, N. & Porter, D. A study of modern linux api usage and compatibility: What to support when you're supporting. *Proceedings Of The Eleventh European Conference On Computer Systems*. pp. 1-16 (2016)
11. Shridhar, M., Adams, B. & Khomh, F. A qualitative analysis of software build system changes and build ownership styles. *Proceedings Of The 8th ACM/IEEE International Symposium On Empirical Software Engineering And Measurement*. pp. 1-10 (2014)
12. Santolucito, M., Zhang, J., Zhai, E., Cito, J. & Piskac, R. Learning CI configuration correctness for early build feedback. *2022 IEEE International Conference On Software Analysis, Evolution And Reengineering (SANER)*. pp. 1006-1017 (2022)

13. Maudoux, G. & Mens, K. Correct, efficient, and tailored: The future of build systems. *IEEE Software*. **35**, 32-37 (2018)

14. Zheng, S., Adams, B. & Hassan, A. Does using Bazel help speed up continuous integration builds?. *Empirical Software Engineering*. **29**, 110

15. Randrianaina, G., Tërnava, X., Khelladi, D. & Acher, M. On the benefits and limits of incremental build of software configurations: an exploratory study. *Proceedings Of The 44th International Conference On Software Engineering*. pp. 1584-1596 (2022)

**Shenyu Zheng** is currently pursuing a Master's degree at Queen's University, focusing on build systems. His research primarily explores the performance and correctness of modern artifact-based build systems, within a CI context. Contact him at 22sz3@queensu.ca

**Dr. Ahmed E. Hassan** is the NSERC/RIM Industrial Research Chair in Software Engineering for Ultra Large Scale systems at Queen's University, Canada. He spearheaded the organization and creation of the Mining Software Repositories (MSR) Conference and its research community. He co-edited special issues of the IEEE Transactions on Software Engineering and the Journal of Empirical Software Engineering on the MSR topic. Early tools and techniques developed by his team are already integrated into products used by millions of users worldwide. His industrial experience includes helping architect the Blackberry wireless platform at RIM, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. He is the named inventor of patents in several jurisdictions around the world, including the United States, Europe, India, Canada, and Japan. He is a member of the IEEE.

**Dr. Bram Adams** is a full professor at Queen's University. His research interests include software release engineering (pre- and post-AI) and mining software repositories. His work has received the 2021 Mining Software Repositories Foundational Contribution Award. In addition to co-organizing the RELENG International Workshop on Release Engineering from 2013 to 2015 (and the 1st/2nd IEEE Software Special Issue on Release Engineering), he co-organized the first editions of the SEMLA event on Software Engineering for Machine Learning Applications. He has been PC co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019, and ICSE 2023 software analytics area co-chair. He is a Senior IEEE Member. Contact him at bram.adams@queensu.ca