Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Measurement

Master's Thesis

Pog: A portable package manager for Windows

Matěj Kafka

kafkamat@fel.cvut.cz
https://github.com/MatejKafka/Pog

Supervisor: Ing. Miroslav Prágl, MBA

Study programme: Open Informatics
Branch of study: Computer Engineering

May 2024

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kafka Matěj**	Personal ID number:	**483777**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Measurement**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Engineering**		

II. Master's thesis details

Master's thesis title in English:

Pog – a portable package manager for Windows

Master's thesis title in Czech:

Pog – správce přenosných balíčků pro Windows

Guidelines:

Many applications for Windows provide a portable version, storing application data in the same directory as the application itself, with minimal alterations to the system. The student will implement a package manager for portable applications.
1. Compare and contrast standard systemwide package installation with portable applications.
2. Analyze popular package managers, both for Windows and other platforms, describe the problems solved, the shortcomings and the features that are typically expected by users.
3. Analyze existing portable applications, describe features required of a package manager to install them.
4. Design and implement a package manager for portable applications, which:
a) accesses an online repository of available portable applications
b) automatically downloads an application and makes it available to the user, behaving similarly to a natively installed application
c) supports uninstalling applications
d) ensures installed applications work correctly when moved to a different machine
5. Document available features of the package manager, including both the user UI and the APIs available to package authors.

Bibliography / sources:

[1] Hisham Muhammad, Lucas C. Villa Real, and Michael Homer. 2019. Taxonomy of Package Management in Programming Languages and Operating Systems. In Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS '19). Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3365137.3365402.
[2] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In Proceedings of the 18th USENIX conference on System administration (LISA '04). USENIX Association, USA, 79–92. https://dl.acm.org/doi/10.5555/1052676.1052686.
[3] Microsoft. Microsoft/winget-CLI: Windows Package Manager CLI (Aka Winget). Microsoft, https://github.com/microsoft/winget-cli.
[4] Chocolatey - the Package Manager for Windows, https://chocolatey.org/.
[5] Scoop, https://scoop.sh/.
[6] Portableapps.com - Portable Software for USB, Portable, and Cloud Drives, https://portableapps.com/.

Name and workplace of master’s thesis supervisor:

Ing. Miroslav Prágl, MBA   Department of Computer Systems FIT

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **08.02.2024**   Deadline for master's thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

Ing. Miroslav Prágl, MBA Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

Abstract

The traditional approach of installing Windows applications system-wide using executable installers leads to inconsistencies, automation issues, and lack of transparency and hinders the implementation of features desired by the developer community. Portable applications avoid many of these issues at the cost of convenience and system integration.

This thesis presents Pog, a new command-line package manager for Windows that fully encapsulates applications in a portable directory, preserving system integration while enabling unprivileged installation, migration between machines, and side-by-side installation of multiple versions. Compared to existing alternatives, Pog also provides significant performance benefits for most common operations.

Abstrakt (CZ)

Tradiční postup instalace aplikací pro Windows do systémových složek, jenž využívá instalačních programů, způsobuje nekonzistence, problémy s automatizací, netransparentnost procesu instalace a brání implementaci funkcí požadovaných vývojářskou komunitou. U přenosných aplikací se většina těchto problémů nevyskytuje, jsou však uživatelsky nepohodlné a neintegrují se do systému.

Tato práce představuje Pog, nový správce balíčků pro Windows, který aplikace plně enkapsuluje v jednom adresáři, zachovává integraci se systémem a zároveň umožňuje neprivilegovanou instalaci, přesouvání mezi systémy a paralelní instalaci více verzí. Ve srovnání s alternativami je Pog také výrazně rychlejší pro většinu obvyklých operací.

Acknowledgement

I would like to thank my supervisor for taking on this thesis and my colleagues Jakub Dupák, Jáchym Herynek, Max Hollmann, and Vojtěch Štěpančík for all of our discussions, mental support, and our shared drive to always do better. Without you, I would not be half the engineer I am today.

I also express heartfelt thanks to Aga, my girlfriend, for her unwavering support during all the hardships in my studies and life outside academia.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 24. 05. 2024

....

Contents

Section 1

Introduction

Linux and other operating systems of the UNIX heritage have a long tradition of using package managers to install and uninstall applications. Package managers simplify many aspects of application distribution, especially due to automatic dependency management and a unified interface for users to discover and install applications. Most Linux package managers are specific to a single distribution, or a family of related distributions, and are a major aspect of what sets them apart. Installation of a package by the package manager primarily consists of extracting a static archive, with limited support for dynamic scripting during the installation.

In comparison, Windows historically did not have a package manager, and most software is still distributed using installers directly from the vendor. This model is more flexible than the one typical for Linux package managers, at the cost of higher complexity and inconsistency both for the developer and the end user. Additionally, since the system cannot reliably track all resources affected by an installer, clean uninstallation of a package is non-trivial. Third-party package managers are available on Windows, such as Chocolatey[1] and Scoop[2], but unlike on Linux, they were not commonly used until recently, when WinGet[3], an official package manager from Microsoft was introduced. Both Chocolatey and WinGet utilize existing installers.

Most Linux package managers install packages system-wide and require root access, and similarly, most Windows installers require administrator rights and install software to the `Program Files` system-wide folder. As an alternative for Windows, there is a tradition of so-called *portable applications*, which do not require installation and store user data and configuration files in the same directory as the program binary. For the end user, however, the current experience of using portable apps is subpar, as the apps do not provide Start menu shortcuts, file association support, `PATH` integration for command line programs, and other features users have come to expect. The PortableApps.com[4] project attempts to remedy some of these issues, but it focuses only on GUI applications, has complex wrapper configuration format, opaque launchers, and no command-line interface.

Pog is an in-development package manager, which attempts to build upon portable applications, providing better system integration and a simple way for developers to author packages and for users to install applications and keep them up-to-date. It is built using a combination of C# and Windows PowerShell 5, with a first-class interactive command-line interface, including full auto-complete. Pog explicitly disallows binary installers, and only supports installation from plain archives and installers that can be extracted without executing any embedded code. For install-time configuration, Pog provides package authors with a comprehensive library of idempotent PowerShell functions to configure the

[1]https://chocolatey.org/

[2]https://scoop.sh/

[3]https://github.com/microsoft/winget-cli

[4]https://portableapps.com/

installed program and export entry points, including Start menu shortcuts and commands in `PATH`. Pog also supports wrapping non-portable applications, as long as they provide a way to configure paths where user data is stored, by creating shim executables that invoke the original binary with modified arguments and environment.

The remainder of this thesis is is organized as follows. Section 2 reviews the history and current state of package management on the three major platforms, with Section 3 establishing the resulting design goals. Section 4 follows with description of options to make existing applications portable. Section 5 gives a high-level overview of Pog, with the following three sections detailing specific parts of the implementation. In Section 9, Pog is shown to meet the specified design goals, followed by a sketch of possible future improvements and a conclusion.

Section 2
Background

There are two common types of package managers, used in different parts of the software distribution process. *System package managers* are a tool for end users to manage installed applications and their dependencies, and tend to be specific to a single platform. In comparison, *language package managers* are typically cross-platform tools, designed to manage build-time dependencies, which are generally private to a single project. For historical reasons, system package managers on Linux are also used as language package managers for C and C++ projects, while some language package managers, such as pip, are also misused as system package managers. The rest of the thesis focuses on system package managers, explicitly ignoring build-time dependencies. I will only call them “package managers” here-on-forth.

Package managers provide a unified user interface for installing and uninstalling applications and their dependencies, ensuring that the system is in a consistent state where all packages have their dependencies installed. During installation, many package managers track the installed files, so that the package may be cleanly removed during uninstallation, and to catch collisions between packages, where multiple packages attempt to write to the same file. Most package managers also support updates, allowing the user to query for new versions of installed packages and install them.

Package managers also aid software discovery, allowing the user to find relevant packages for their use case based on name, tags or description. Most package managers have maintainers who actively adapt and curate the available software, removing malware and abandonware and enforcing consistent conventions around installation locations and package quality and providing a central point of support for users.

The feature set of different package managers varies widely. Some package managers require specifically built packages from the source, while others adapt package-independent archives or binaries. Some install packages system-wide, requiring administrative privilege, while others install packages privately for a single user. Some use a custom directory structure; others follow the platform conventions. Some allow packages to script arbitrary changes to the system; others only extract a static archive. Some can build and manage arbitrary dependencies; others only install self-contained packages. Some support installing many versions of a package, while others offer multiple versions in a repository while only supporting the installation of a single version at a time, yet others only provide the latest version of each package. Some only offer a command line interface, while others also provide a graphical interface.

The following section summarizes the history and current state of package management on the three major contemporary desktop platforms and discuss how the platform influences the above trade-offs.

2.1 Linux package management

Before the advent of build tools, software on UNIX-like systems was distributed in the form of source code with compile scripts. After Make was developed in 1976, a large fraction of existing software converged on Make, built and installed by invoking `make && make install`. However, UNIX systems varied widely, and each project had to check compatibility with various targets. To ease the burden, GNU Autoconf[5] was released in 1991 as a tool for generating a platform-specific Makefile based on a series of feature tests. However, this still required the user to build all installed software locally.

In 1993, Slackware's pkgtools were released as the first popular package installer, where the end user did not have to build the software. Instead, a package maintainer built the packaged software and packed the output as a tarball, which was distributed to the users. The end user could either unpack the tarball to the root of the filesystem manually, or use pkgtools, which unpacked the tarball and stored a list of the unpacked files, which was used to later cleanly remove the package.

In the following years, other package managers for new Linux distributions, such as Debian's `dpkg` and `pms` from Bogus Linux added package metadata to the tarballs, including a list of dependencies, and warned the user when they installed a package with unmet dependencies, resulting in the modern concept of a package. The packaging aspect of the issue was mostly solved, but packages still had to be discovered and downloaded manually.

Meanwhile, FreeBSD was also released in 1993, taking a different approach with ports trees. A ports tree was a directory hierarchy of available packages, called *ports*, where each package directory only contained a Makefile and optionally a set of patches. The Makefile could automatically download the source archives, apply patches, build the software and install it. Since the ports tree was tiny compared to the actual source archives, it could be reasonably stored locally and kept up-to-date. To install a port, the user ran `make install` inside the port directory. This was likely the first well-known package installer system using a package repository.

The two branches converged over the following years, with various Linux distributions implementing support for package repositories by expanding upon existing local package installers, and FreeBSD starting to use binary packages. A notable early example is the `dselect` tool for Debian, implemented as a `dpkg` front-end, with support for remote package repositories added in 1997, and the well-known `apt-get` package manager, also for Debian, released in 1998.

2.1.1 Contemporary package managers

Today, there are six major families of desktop package managers, listed in Table 1. All major desktop Linux distributions use a package manager to service most installed software, including the kernel and other core components. Packages are installed system-wide to standard directories and root privileges are required to manage them. Software must be packaged for a specific package manager and typically built from source by the pack-

[5]https://www.gnu.org/software/autoconf/

Distribution	Frontend	Engine
Debian	APT	dpkg
Ubuntu		
Mint Linux		
Pop!_OS		
Fedora	YUM DNF	rpm
CentOS		
RHEL		
SUSE Linux	Zypper	
openSUSE		
Arch Linux	pacman	
Manjaro		
Gentoo	Portage	
NixOS	Nix	

Table 1: Well-known Linux distributions and their native package managers.

age maintainer, instead of repackaging existing binary releases. Since packages use fixed paths, only a single version of a package may be installed at a time, and package maintainers typically depend on the latest version of each dependency. In cases where multiple versions of a package are needed, such as for major versions with breaking changes, a separate package is kept for each version. Local copies of package listings are kept for each package repository to speed up operations and must be manually updated by the user.

There are a few notable exceptions. The first one is Nix [1], [2], the native package manager for NixOS, which also supports most Linux distributions and macOS. Nix attempts to ensure reproducibility and isolation between packages by compiling packages in an isolated environment, where only explicitly declared dependencies are present, installing each package to a separate directory named using a hash of all sources of the package, and using package-specific shared library paths. This allows Nix to trivially install multiple versions of a package without conflicts and also enables non-privileged users to install packages without affecting other users. Closed-source packages are built by repackaging binary releases and patching library paths. To make binaries accessible to the user, Nix creates symbolic links in a per-user directory, which the user may add to the `PATH` environment variable.

Another well-known distribution-agnostic package manager is Flatpak[6], managing self-contained, sandboxed packages that bundle all dependencies. Flatpak packages may be installed by unprivileged users, however, no support for installing multiple versions of a package is provided.

[6]https://flatpak.org/

A similar design space is explored by the AppImage[7] project. AppImage is a tool to create self-contained portable packages, which can be directly launched without an installation step. The package is an ELF binary, which acts as a launcher and contains an embedded SquashFS [3] image, which is mounted by the launcher and contains the actual packaged application. This allows the user to easily install multiple versions just by downloading multiple AppImage binaries. Applications are not sandboxed and store data in standard paths.

2.2 macOS package management

Author's note: My personal experience with macOS is fairly minimal. From my research, there are only a few aspects of application management on macOS that are potentially relevant for the topic of this thesis, described below.

Unlike most desktop Linux distributions, macOS does not have an official package manager outside of the App Store, which is not widely used. Graphical applications are stored in the `/Applications` directory as subdirectories with an `.app` extension, which are treated by the system user interface as a single executable file. To manually install an application, the user downloads an archive containing the application and copies the application to the `/Applications` directory; the system automatically detects the newly added directory and registers it based on a declarative `.plist` configuration file stored inside the application directory. More complex configuration, such as registering daemons, is typically done by the application on first launch.

To uninstall an application, the user moves the application `.app` directory to Trash. In recent versions of macOS, the system attempts to automatically clean up daemons registered by the application; however, it is generally the responsibility of the application to clean up after itself, often unfulfilled.

For power users, there are two popular third-party package managers – Homebrew and MacPorts. MacPorts is heavily inspired by the FreeBSD ports system and focuses on building packages from source. Homebrew supports both open-source packages built from source and closed-source software packaged from a binary release; for pre-built packages, it operates similarly to mainstream Linux package managers described in the previous section.

2.3 Windows package management

Similarly to macOS, Windows did not have an official package manager for most of its history and users downloaded applications directly from the vendor. However, Windows does not have a self-contained app format like the macOS `.app` bundle, making installation significantly more complex. Each feature the application supports, such as Start menu invocations, file associations, system services and others must be configured separately, in varying parts of the filesystem and Windows Registry. To set up the application, vendors typically provide an installer – an executable file that installs the application in any way the vendor sees fit.

[7]https://appimage.org/

This model is very flexible, at the cost of inconsistencies in the installation process, inscrutability for the user, less centralized oversight over application quality and additional work for the vendor, who must create the installer. The last point is especially problematic since for many vendors, creating an installer is an afterthought, resulting in unwillingness to learn and invest in the process, and subsequent low quality of the installers produced.

There are a few common approaches to building installers. The least complex option is a self-extracting archive, an executable file that contains a static archive and a small program that which presents a simple UI to the user and extracts the archive to a selected location. Some formats also support other installation steps, such as creating a Start menu shortcut or registering the application uninstaller. The archive format used, such as ZIP, typically allows prepending extra data to the archive, meaning that a self-extracting archive is also a valid static archive. The upside of this approach is that the installer is very simple to create and inspect, at the cost of support for more advanced configuration and scripting, making this approach suitable only for simple applications.

Another common option is using third-party executable installers, such as NSIS[8] and Inno Setup[9], which are conceptually similar to self-extracting archives while also providing extensive support for installing common application features and custom scripting. Many implementations provide a simple graphical UI to create an installation package with minimal knowledge. However, there are multiple issues for the user. The installers tend to be hard to inspect and modify. Automating the installation tends to be non-trivial since the support for silent installation and configuration varies widely. If the installation is interrupted, the application may be left in a partially installed state, and no support for repairing an existing installation is typically provided.

2.3.1 Windows Installer

To alleviate these issues, Microsoft created Windows Installer [4], an installation service built into Windows that operates on installation packages in the MSI format. Each package is a declarative database containing a sequence of high-level actions needed to install, repair or uninstall the package and the resources necessary, such as the installed files, registry entries, and shortcuts. [5] An extensive library of built-in actions is provided; a vendor may implement custom actions if necessary, although this is generally not recommended. Except for custom actions, the format is fully inspectable with commonly available tools and may be freely customized before installation using *transforms*. [6] Windows Installer supports reliable silent installation, extraction of the contained files, and provides an extensive logging system. [7]

During installation of an MSI package, Windows Installer stores a log of all changes to the system. Coupled with the use of restore points and NTFS and registry transactions [8], this ensures that even when the installation is interrupted, the system can roll back to a consistent state. A database of all installed packages and their components is kept, which allows easy auditing of installed software. In domain networks, Microsoft provides

[8]https://nsis.sourceforge.io/Main_Page

[9]https://jrsoftware.org/isinfo.php

extensive tooling for seamless centralized installation using Group Policy, including on-demand installation of packages on first use.

The extensive feature set and consistency checks results in high complexity for vendors and users and slower installation speed compared to alternatives. Additionally, Microsoft does not provide any graphical tool to author MSI packages, only the XML-based WiX[10] toolset, which closely follows the internal database structure of the MSI format. Third-party MSI tooling, such as InstallShield[11], is generally more complex to use and expensive to license compared to executable installers. As a result, many vendors still use other installation options.

2.3.2 Universal Windows Platform

With Windows 10, Microsoft introduced the Universal Windows Platform [9] (UWP), with sandboxed applications, a new system API and runtime, and a new fully declarative MSIX package format, with no support for custom scripting. UWP applications are distributed primarily through the Microsoft Store, installation from other sources is disabled by default. Applications do not have access to the filesystem unless explicitly allowed by the user, except for application data directories. Many parts of the UWP platform are proprietary, tied to a Microsoft account, with minimal support for tweaking and backing up applications, and many of the isolation primitives are under-documented, or not documented at all. Due to various limitations, application vendors are generally reluctant to use the platform and most software is still distributed using the previously described methods.

2.3.3 Portable applications

To avoid the complexity of creating an installer, some projects instead distribute static, self-contained archives, which the user manually unpacks to a custom directory, where the application may also store application data. A significant upside of this approach is that the application may be freely moved between different machines while preserving configuration, and that multiple versions of an application may be installed side-by-side. Support for multiple versions is especially useful for runtime environments; each popular runtime already has a separate tool to switch between versions, such as `rbenv`[12] for Ruby and `nvm`[13] for Node.js, which internally use a portable version of the runtime.

Since portable applications generally do not depend on the state of the system, they tend to be more reliable compared to more deeply integrated applications. However, the downside is a significantly degraded user experience – the user must manually unpack the archive, the application must be launched by navigating to the directory instead of using the Start menu, and absolute paths must be used to invoke the application from the command line. Applications that register services or scheduled tasks, provide shell extensions, or otherwise integrate with the system are hard to distribute in a portable format.

[10]https://wixtoolset.org/

[11]https://www.revenera.com/install/products/installshield

[12]https://github.com/rbenv/rbenv

[13]https://github.com/coreybutler/nvm-windows

Multiple projects attempt to streamline the experience of using portable applications, most notably PortableApps.com[14], which provides a packaging system and a graphical interface to install, run, and maintain portable applications. Each application is packaged in a standalone package, which may be installed without using the graphical interface. Each application directory contains an executable launcher generated during packaging using a configuration file and an NSIS script. The launcher is opaque, with no support for introspection or modifications after packaging. There is no command-line version of the graphical interface, and no command-line applications are provided in the official repository.

2.3.4 Package managers

Unlike on Linux, where most commonly used software is open source, proprietary closed-source software has been the norm on Windows, only slowly changing in the last few years. This is a significant obstacle in implementing a Linux-style package manager since closed-source software cannot be easily adapted for packaging, and most proprietary software cannot be legally redistributed. Instead, all well-known package managers use binary releases provided directly by the vendor, and the package itself only describes how to retrieve and install the release from a third-party server.

2.3.4.1 Chocolatey

The first popular third-party package manager for Windows was Chocolatey [10], released in 2011. Since most applications already have an installer, Chocolatey downloads and runs the existing installer and configures some of the exposed options, requesting silent installation. Portable applications are downloaded and unpacked by Chocolatey to a configurable directory. Non-administrative installation mode is supported for portable applications. To make binaries available on `PATH`, .NET-based executable shims (Section 8) that invoke the application binary are placed in a single public directory.

Each package contains a PowerShell script that installs the application using a library of high-level functions provided by the run time environment. However, since the script must wrap an existing installer and conditionally pass various options based on package parameters, it tends to be somewhat verbose. [11] As Chocolatey does not control the installation process, installs packages system-wide and does not reliably disable auto-update, the state of the system may somewhat easily become inconsistent with the metadata that Chocolatey internally tracks.

Chocolatey utilizes the package and repository format of NuGet, the language package manager for .NET, and internally uses the NuGet client to interact with repositories. Multiple versions of a package are available from the repositories; however, installing multiple versions side-by-side is not well-supported, especially for installer-based packages and has been deprecated in 2022. Packages may have dependencies on specific versions, but until 2022, dependency versions conflicts were not handled, and conflicting dependencies were silently overwritten. [12]

[14]https://portableapps.com/

2.3.4.2 WinGet

In 2020, Microsoft released Windows Package Manager [13] (*WinGet*), *strongly inspired* [14] by the since discontinued AppGet[15] project. Similarly to Chocolatey, it reuses existing installers and supports installation of portable applications. However, it integrates well with existing Windows infrastructure, registering even portable applications for uninstallation and using the registry as a source of truth about installed applications. As a result, applications installed by WinGet may be seamlessly managed using the built-in Windows user interface and in turn, manually installed applications can be managed using WinGet. Non-administrative installation is supported for most, but not all applications, depending on the installer type.

WinGet supports installation of both Win32 applications from dedicated Git-based repositories and UWP applications from Microsoft Store. The package manifests are written in declarative YAML (see Appendix D) with no support for scripting. Multiple versions of each package are available, but only a single one may be installed at a time. Packages may specify dependencies, optionally setting a minimum required version; the dependency system tends to be used for application-level dependencies, and libraries are still bundled in most, if not all, installers.

Since its release, WinGet has achieved widespread adoption among developers as the official Windows package manager, likely saturating the market for system-wide package managers. However, it is still limited by its reliance on existing installers and their inconsistencies and does not significantly change the authoring experience for software vendors.

2.3.4.3 Scoop

Motivated by the complexity of existing installers, the Scoop [15] project, started in 2013 and officially released in 2022, takes inspiration from Linux package managers. Instead of running an existing installer, Scoop installs applications by unpacking static archives or installers in supported formats without executing untrusted code. By default, the packages are unpacked into a per-user directory without a need for administrator rights. For portable applications, Scoop persists portable data directories between updates using junctions or symbolic links. [16] Non-portable applications store data at default paths.

System integration of installed applications is limited to adding Start menu shortcuts, exporting binaries to `PATH` using a .NET-based executable shim (see Section 8), and adding custom subdirectories to `PATH` for applications which dynamically add executables at run time, such as language package managers. Package manifests may specify a PowerShell script that is executed after installation to further configure the application where necessary. While this restricts potentially useful application features, the upside is a simpler, faster and more transparent installation experience. Scoop uses the on-disk state of installed packages as the source of truth instead of keeping a separate database of package metadata, sidestepping any potential consistency issues.

Scoop package repositories are Git-based and use simple declarative package manifests written in JSON (see Appendix C). Only the latest version of each package has a manifest.

[15]https://appget.net/

To install older versions, Scoop uses a built-in auto-update mechanism to heuristically derive an older version of a manifest; in practice, the heuristics tend to fail for older package versions. Scoop installs each version into a separate subdirectory, allowing installation of multiple versions side-by-side and easy switching by redirecting a single junction. Packages may specify application dependencies by name but not by version; library dependencies are not supported. [17]

While Scoop is fully implemented in PowerShell, it does not provide integration and intentionally deviates from PowerShell conventions. [18] The implementation is a collection of shell scripts with minimal encapsulation and utilization of the object-oriented nature of PowerShell. There are only minimal attempts at ensuring consistency in case of an interrupted installation and other unusual circumstances. Nevertheless, the user experience is surprisingly pleasant for typical use cases, both for the end user and package author.

Section 3

Design goals for Pog

The previous section provided an overview of the current state of package management on Windows. For users seeking to streamline the installation process of system-wide applications, WinGet is a viable, officially supported option. However, the quality and transparency of the application installers used by WinGet vary widely, and installation speed is often subpar. Additionally, creating the installer is an unnecessarily complex step for the vendor.

Using system-wide applications raises other issues described in the previous section. Portable applications present an alternative option, which resolves most of the issues with system-wide installation, and dramatically simplifies the release process for vendors. Scoop successfully explores the idea of installing applications without using an installer, but does not attempt to encapsulate the applications and provides a limited support for installing multiple versions or moving applications between machines.

The primary motivation behind Pog is to explore the feasibility of building a package manager that fully encapsulates applications within a single portable directory, supporting unprivileged installation, migration between machines, and side-by-side installation of multiple versions. The installation process should be fully transparent for users and accessible for package authors, encouraging developers to use Pog even during local development. The rest of this section details the core design goals.

3.1 Encapsulation and portability

Installed applications should not be bound to a single computer and Windows installation. It must be possible to move the application to a different machine, or reinstall the system, while keeping all data intact and the application correctly working. Where possible, system should not need to know about the installed applications.

With the current model of installation on Windows, where an application is split across multiple folders, this is hard to implement. The current model has the advantages of easier permission control, multi-user support and for domain networks, the possibility of having binaries and data on separate network drives. However, for personal computers, which typically only have a single user, the latter two are not very valuable in practice, and even permission control is of limited use, typically only utilized to prevent unprivileged programs from modifying the binaries of other installed applications. In practice, since the primary target of most attacks is acquiring user data or gaining persistence, both of which are trivial even without administrative access, the described security boundary is not too valuable. Furthermore, an increasing number of applications installs binaries to the `AppData` directory, precisely due to lower privilege requirements. Even on shared servers, the long-term trend seems to be moving towards virtual machines and containers, both of which typically only have a single user account.

Thus, in Pog, each package should store both application files and data in a single directory, which can be moved without breaking the application or losing data. This also prevents file conflicts between packages, provides a single location where to look for any relevant configuration for each package, and simplifies uninstallation. All packages should have the same directory structure for application files, data files, caches, and logs to simplify backup and other automated processing of the packages.

3.2 Minimal external dependencies

Expanding upon the principle of encapsulation, packages should minimize external dependencies. It is the responsibility of the package author to acquire all runtimes and libraries required for correct functionality.

On other platforms, it is common for applications to have many dependencies, and considered a good practice to not include any third party libraries in a package directly. This leads to smaller package sizes and faster roll out for bug fixes in libraries. The downsides are that the package manager must implement dependency resolution, dependency version conflicts between packages are possible (unless multiple versions of a package can be installed, which is not the case for most package managers), higher error surface and lower performance due to dynamic linkage.

On Windows, since there's no shared repository of packages to depend on, most applications ship all of their dependencies. This leads to larger package sizes, but greatly simplifies package management and testing. Additionally, with link-time and profile-guided optimization added to compilers, there's a larger performance incentive for static linkage, which works well with this approach; this is one of the reasons why many modern programming languages, such as Go and Rust, default to static linkage.

Another advantage of bundling dependencies is the possibility of effortlessly installing multiple versions of a package side-by-side. This is especially desirable for runtime environments and software compilation and testing. Pog should be able to install and run multiple versions of an application side-by-side and properly isolate data and configuration used by each version.

3.3 Transparent installation

User should be able to easily check what changes will be done during package installation, and have the freedom to modify the package. No closed-source code should execute during package installation, and most of the functionality should be provided by the package manager, not by an installer script. This goal specifically precludes installing a package using an installer.

The installation process should be described in a plain-text script, with sufficiently high-level actions to make it easily readable for curious users. The scripting language used should be well-known and supported; learning a new language for package scripting is not acceptable, since most users and authors do not interact with the scripts often enough to make it worth their while.

3.4 Accessibility

Pog should be a pragmatic tool, accessible to users unwilling to invest time in in-depth study; accessibility should be a priority over *theoretically correct*, but harder to use designs.

As much as possible, Pog should use the actual on-disk state of a package as the source of truth, instead of keeping an internal representation of the package state. As a result, it should be hard to introduce inconsistencies by manually modifying the package directory and easy to restore the package to a valid state, encouraging tinkering.

Furthermore, Pog should be scriptable, providing a consistent interface to inspect package state and metadata and add custom functionality not present out-of-the-box, allowing users to build upon Pog.

3.5 Packaging experience

Creating Pog packages should be easy, even without any prior experience or external tooling. The package manifest should use a standard, well-known format and scripting language. Anyone should be able to create a package, without cooperation from the original author and access to the source code. Additionally, Pog should be useful as a tool for local development and integrating separately downloaded programs with the system; using a local-only package should be trivial.

Since the packaged software must be adapted to function as a portable application and follow the package conventions, Pog should provide a straightforward and flexible way to configure the package, with shell scripts being an obvious choice. To ease authoring of packages, a high level library of well-tested, consistent functions should be provided; in the author's view, experience from other installer scripting environments shows that if the provided configuration primitives are too low-level, package manifests become hard to read and many minor inconsistencies between packages are introduced.

Section 4

Making applications portable

Windows-native software tends to store user data in `AppData` and `ProgramData`. POSIX-native software ported to Windows often uses the user's home folder instead. To make an application portable, it must be configured to store user data at a custom path. A minority of applications explicitly support portable mode if configured to do so. Many other applications have a way to override the default paths to aid testing during development, which can also be used for implementing portable mode.

A second obstacle for managing portable applications using a package manager are automatic updates. Many end-user applications check for new updates at launch, and automatically install the update. For an application managed by a package manager, this is undesirable – the package version will not match the application version, resulting in a confusing user experience and issues with reproducibility, while also slowing down application start up. Additionally, a user may explicitly install an older version of a package, in which case automatic updates would be counterproductive. Most such applications provide a way to disable automatic updates. Both the data paths and automatic updates can be typically configured using one of the ways described in the following sections.

4.1 Environment variables

Configuration through environment variables tends to be the most convenient option for packaging portable applications. No modification of the wrapped application is required, and the variables are automatically propagated to child processes, which ensures that applications spawning child processes or restarting the main process keep the desired configuration. Additionally, unlike command line arguments, environment variables are parsed consistently in all applications, which makes support for user overrides trivial – if the configuration environment variable is already set by the caller, the launcher may leave the variable untouched. However, supporting user overrides is somewhat risky in cases where a system-wide installation of an application may also set the environment variable globally (e.g. `JAVA_HOME`), resulting in the supposedly portable wrapped application accessing user-wide data.

There are two other notable downsides, the first one is the need to use a custom launcher to configure the environment variable before invoking the wrapped application. The second downside is that if a standard environment variable like `%USERPROFILE%` is modified in the launcher, and the wrapped application launches an application from another package or an external application, it inherits the changed environment and may behave incorrectly. For packaged applications, this tends to be a non-issue, as they should not access any user-wide paths, but issues arise when invoking a non-packaged application. Fortunately, in practice, most applications either use a custom environment variable, or do not invoke other programs.

4.2 Command line arguments

Configuration through command line arguments also avoids modifying the wrapped application. Additionally, Windows shortcut files (`.lnk`) support passing extra arguments, thus it would seem no extra launcher is needed for GUI applications. However, since the Win32 `CreateProcess` function cannot invoke shortcuts, Windows copies the target command line from the shortcut when a user sets up a file association using the *Open with* menu or pins the shortcut to the Start menu and invokes the target directly instead of invoking the shortcut. This becomes an issue when a subsequent package update modifies the shortcut, and the copies become stale. For this reason, using a launcher is generally necessary both for console and GUI applications.

Unlike with environment variables, the application must take care to explicitly forward any relevant command line arguments when invoking child processes or restarting the main process. The upside is that the application may safely invoke other applications without the risk of polluting their environment variables.

Another major downside is that command line parsing is done by the application, and there's a considerable variance between applications in what counts as an argument separator, how are parameter names denoted (`-`, `--` and `/` prefixes are all commonly used) and how parameter values are passed (either as a next whitespace-delimited token, or separated by `=` or `:`). As long as the launcher only needs to prepend or append arguments to the command line, most of the complexity can be avoided. However, supporting user overrides tends to be tricky, unless the application already supports overriding, such as by using the last provided value for named parameter and ignoring previous duplicates.

A related issue is presented by applications with strict requirements on the command line structure. This is most common for applications with the `<command> <subcommand>` argument structure, such as Git, which often only accept some arguments before the subcommand and other arguments after. Similar issue occurs for applications which only accept parameters configuring portable data paths for sub-commands which use them, but not others, such as accepting `<command> --cache-dir=<path> run`, but not `<command> --cache-dir=<path> version`. For both of these cases, the launcher has to have some awareness of the allowed argument structure to correctly parse the provided command line and insert arguments, significantly complicating packaging.

4.3 Configuration files

Some applications detect the presence of a configuration file or directory in the same directory as the application binaries and use it instead of the default data paths. The upside of this approach is that configuration is trivial for users who are manually installing a portable application, since no custom launcher is needed, making it the most common implementation for applications that explicitly offer portable mode.

The main issue of this approach tends to be that the portable data paths are not configurable. This is an obstacle both to a consistent package structure and to reusing a single binary directory between multiple configurations of the app. To force the application to write to a specific portable directory, different from the default one usually requires using a symbolic link or a junction, the downsides of which are explored in Section 6.3.2

and Section 8. Some applications solve this issue by also looking for a specifically named file containing the desired path to the data directory, and use that instead of the default portable data path, which avoids the need for linking the directory, but still precludes using multiple configurations with a single installation of the application.

Another somewhat common approach is using one of the previous methods to set a custom main configuration file path, which contains other data paths. This resolves the aforementioned issue, at the cost of requiring modifications to the configuration file whenever the application directory is moved, which requires a parser for the file format used.

4.4 Sandboxing

An alternative approach to convincing the application to voluntarily use portable data paths is to intercept access to the filesystem and registry and redirect it to a custom location. This is the approach taken by Microsoft for virtualized UWP apps [19] and other parts of Windows, and by third party tools such as Sandboxie[16].

The upside of this approach is the seemingly minimal application-specific knowledge or cooperation from the application required, making automated packaging possible. Unlike the previously described approaches, redirection will not break if an application adds a new data path in an update.

The most severe issue is the complexity of implementing a robust sandbox – it needs to correctly redirect all possible avenues of access to the filesystem and registry, without significantly impacting performance of sandboxed applications. This also entails tracking and sandboxing launched child processes, since many applications use internal subprocesses as part of normal operation, while correctly detecting sub-processes that do not belong to the application – as an example, a shell sub-process spawned by a terminal emulator should be excluded from the sandbox, but a shell sub-process spawned to launch a command on behalf of the application should be sandboxed. A related issue is an application accessing resources using an out-of-process COM service, however a niche scenario. However, if some of the requirements are relaxed, it should be feasible to author a sandbox capturing the relevant operations for most common applications, which could still prove a valuable packaging tool.

Another, less severe issue with this approach is that the sandbox cannot distinguish between operations done by the application while accessing the data paths, and operations done on the behalf of a user. This could be an issue for applications such as a text editor, where the user would not be able to access files under `AppData`.

There are two common implementations of sandboxing. One option is using a kernel filter driver, which will intercept system calls from the wrapped application and redirect relevant operations. If implemented well, this approach should be resistant to tampering, and needs to cover a much smaller API surface than the alternative. However, the need to load a kernel driver is a major obstacle, since portable applications are often used on machines without administrator access.

[16]https://github.com/sandboxie-plus/Sandboxie

The second common implementation injects a dynamic library into the application process, which replaces existing system library functions with wrappers, which redirect relevant operations and call the original functions internally. [20] There are two obvious targets – the public Win32 API, implemented primarily by `kernel32.dll`, or the undocumented `ntdll.dll` library, which is internally used by all subsystem libraries and performs the actual system calls. This approach is suitable for portable applications, since no system-wide modifications are necessary. The downside is that the Win32 API has a wide surface, and intercepting all common operations which may result in filesystem or registry access entails a lot of work. Another issue is that platforms such as .NET often access the Win32 API dynamically using `LoadLibrary` and `GetProcAddress`, which bypass the injected library unless it also correctly intercepts these functions.

Both approaches to implementation may be combined, by blocking all potentially problematic operations using a kernel driver, and using an injected dynamic library to selectively enable some functionality by redirecting the calls to a custom driver, as implemented in Sandboxie. [21]

Section 5

The Pog package manager

Pog is a new system package manager for Windows, running in PowerShell 5. All packages are installed by extracting a static archive without external dependencies. Installed packages store data directly in the package directory. This ensures that packages do not interfere with one another and the underlying operating system. Package configuration scripts are executed in a consistent, restricted environment and use a set of idempotent commands to configure the installed package.

Each package has a manifest, which contains the package metadata, describes where the package contents are downloaded from and how to configure the package after installation. Available manifests are retrieved from repositories – Pog supports both local repositories and remote repositories, operating directly on the repository without any intermediate update steps. Packages are installed in a package root, an ordinary directory which may be placed at a local drive, a flash drive or a network share. Multiple package roots may be defined; for example, one package root for packages used on a single machine and another package root on a flash drive for packages used on multiple machines.

Pog is implemented as a PowerShell module. All public commands are documented using the PowerShell help system. Completions are provided for all common parameters, including remote package names and versions. Commands operate on structured .NET types, allowing for introspection and simple pipelining of multiple compatible commands similarly to built-in PowerShell commands, as illustrated in Listing 1.

Originally, in 2020, Pog started as a pure PowerShell Core project. However, over time, the code base became harder to manage, and Pog needed access to low-level Win32 functions to implement features such as atomic installation, which are non-trivial to use from PowerShell, but easily accessible from C#. As a result, most of the core logic is now rewritten as a C# library, which is internally used by the public commands.

Due to short support period for new .NET releases upon which the newer PowerShell Core is built, Windows still ships with Windows PowerShell 5, which lacks some features from newer PowerShell versions. Therefore, during 2023, the code base was migrated to target PowerShell 5, which should be present by default in all supported Windows versions, unless explicitly blocked by the system administrator.

The following sections describe how Pog installs packages, how it uses package repositories and how package manifests are authored.

```
> Find-PogPackage aria2 -AllVersions | select -First 2 `
>   | select Version, {$_.Manifest.Install.SourceUrl}
Version $_.Manifest.Install.SourceUrl
------- -----------------------------
1.37.0  https://github.com/.../aria2-1.37.0-win-64bit-build1.zip
1.36.0  https://github.com/.../aria2-1.36.0-win-64bit-build1.zip
```

Listing 1: Example of the Pog scripting interface.

Section 6
Package installation process

The installation process of a Pog package consists of four steps. First, the package manifest is *imported* by downloading it from the repository to a package directory. Next, the source archives requested in the manifest are downloaded and *installed*, replacing any previously installed version. Afterwards, the package configuration script is executed, *enabling* the package, which sets up the application to run in portable mode and defines entry points to the package, such as desktop shortcuts and console commands. As the final step, the entry points are *exported* and become accessible to the user – desktop shortcuts are copied to the Start menu, commands are exported to a directory on `PATH`.

To install a package available in the repository, the user invokes the `pog` command, which executes all of the four steps in order. For more advanced use cases, the steps may be invoked separately, enabling useful scenarios such as:

- Custom packages, which are not available from a repository. (*install*, *enable*, *export*)
- Wrapping a local project in a Pog package. This is useful for local administrative tasks and custom applications built from source. (*enable*, *export*)
- Packages, which are only accessible by full path, without exporting the entry points; these are useful to resolve entry point conflicts when multiple versions of a package are installed. (*import*, *install*, *enable*)

Crucially, separating the *install* and *enable* steps is what allows Pog packages to work portably from changing paths. The package is only installed once, but the *enable* step can be executed whenever the package directory changes and reconfigures the application for the new path. In the following sections, each of the four steps is described in more detail.

6.1 Importing a package

Package installation starts by retrieving the package manifest from a repository. Package manifests are PowerShell data files [22], a JSON-like format with support for inline functions, illustrated in Appendix B. This format is used to simplify embedding configuration scripts directly into the manifest while still providing good editing experience; in comparison, Scoop manifests embed PowerShell fragments as JSON strings, which simplifies tooling, at the cost of less convenient editing. Additionally, a manifest may also bundle supplementary files, for example batch files used as launchers for the packaged application. The retrieved manifest is placed into the package directory, where the package is then installed.

Pog supports both local and remote repositories. Local repositories are a simple two-level directory structure of packages and available versions of each package, containing the package manifests as leafs. Remote repositories follow a similar structure, exposed over HTTP from a static web server. Repositories are described in more detail in Section 7.

A package may be installed multiple times to use different versions or different configurations side-by-side. To distinguish between the installations, each one must have a unique identifier, which is used as the name of the package directory. When installing a package,

```
Install = @(
  @{
    Url = "https://github.com/mesonbuild/meson/.../meson-0.63.3.tar.gz"
    Hash = '519C0932...'
    Target = "meson"
  }
  @{
    # Meson needs python, install the embeddable zip version
    Url = "https://www.python.org/.../python-3.11.5-embed-amd64.zip"
    Hash = "D82391A2..."
    Target = "python"
  }
)
```

Listing 2: Manifest installation sources

the identifier is configured using the `-TargetName` parameter. Additionally, the user may also select a package root to install the package to using the `-TargetPackageRoot` parameter.

6.2 Installation

Next step is downloading and extracting any source archives specific in the manifest. From here on, only the imported manifest is used and the repository is no longer referenced.

Each package manifest references one or more source archives, which are downloaded and extracted to an `./app` subdirectory of the package. As shown in Listing 2, the manifest contains the source URL and a SHA-256 digest of the file to ensure reproducibility and detect tampering. It may also optionally specify a subdirectory of the retrieved archive to extract, a target subdirectory of `./app` where the archive is extracted or a script to modify or rename the extracted files, such as in cases where the contained binaries use a version suffix.

The archives are downloaded using the BITS [23] service, a built-in Windows download service with good throughput and support for background transfers without impacting interactive usage. To simplify the package structure and user interface, each imported package only have a single version installed. However, fast version switching is desirable for users who alternate between multiple versions of a package during software development and testing, especially for runtimes such as Node.js or Python, and redownloading the package each time would be wasteful.

To resolve this issue, Pog uses a concurrent download cache, where the downloaded archives are stored, keyed by the digest specified in the manifest. Before downloading the archive, Pog checks if the archive is already cached, in which case it uses the cached copy. The download cache is a filesystem directory, which contains a subdirectory for each entry, named according to the entry key. The entry subdirectory contains the cached file, and a metadata file in the JSON Lines [24] format, which lists all packages that used the entry.

To add a new entry, the file is first placed in a temporary directory on the same partition as the cache and the metadata file is added. Then, the directory is atomically moved [25] to the cache directory, while holding a file lock on the entry with shared read access to prevent concurrent deletion. When retrieving an entry, the calling package is atomically added to the metadata file and the entry is locked for reading using a shared read file lock. In both cases, the file lock is held while the cache entry is in use and released afterwards to signal that the entry may now be deleted. The locking protocol ensures that the cache may be used from multiple Pog instances concurrently, and even correctly cleared, skipping entries in use.

The retrieved archives are extracted into a temporary directory inside the package. After the new app directory is ready, it is atomically exchanged with the previous app directory, if it exists. This ensures that even in the event of a power loss or system failure, the package is left in a consistent state.

Unlike POSIX systems, Windows have mandatory file locks, and a process holds a mandatory file lock over its executable and loaded shared libraries, which complicates in-place updates, since it is not possible to replace the app directory while the application is running. Even on POSIX systems, updating application files while the application is running may cause breakage when the old application process tries to access a data file which was changed during the update, or when a new version of the binary is launched concurrently with the old version.

One possible approach to resolve some of these issues, used by Scoop, is to install each version into a separate directory and then redirect callers to the latest versions, either using a junction or a shim executable. However, this may still result in multiple versions of the application operating on the same data, unless the application uses a lock file to detect the situation.

Pog uses a single unversioned app directory, and instead detects that the app directory is in use during installation, in which case it attempts to list the processes locking files in the directory, shows a message to the user prompting to stop the processes and waits for confirmation before continuing.

6.3 Package configuration

After the package is installed, the application typically needs to be configured to run in portable mode, using one of the methods described in Section 4. Additionally, entry points to the application must be exported from the package. Since the process tends to significantly vary between packages, the manifest provides a PowerShell script, which is executed after installation and configures the package. This step is called “enabling” the package.

Each package is enabled after installation, and reenabled whenever the package is moved to a different machine or filesystem location, to reconfigure it for the new environment. It should be safe to rerun the configuration script multiple times, the script should be idempotent.

PowerShell users may customize the shell environment, defining new functions, aliasing existing commands or defining variables which influence how PowerShell evaluates scripts. It is undesirable to run the manifest scripts in the original PowerShell session, since a script may be affected by these modifications and behave differently on different machines. Pog runs the script in a new runspace [26] in the same process, with disabled module autoloading and only a few standard modules loaded. The container runspace ensures a consistent environment, while still allowing Pog to pass live .NET objects between the user session and the runspace.

Configuration of most packages consists of only a few different operations – creating configuration files and directories, creating symbolic links inside the package, configuring properties of executables such as display scaling, and exposing entry points to the application. However, all of the operations have high internal complexity, and implementing them correctly and consistently while authoring the manifest is non-trivial. To improve the authoring experience, Pog provides a library of commonly used functions, which are automatically loaded to the environment where the configuration script is executed. All of the provided functions are idempotent, and log all changes made to ensure the installation process is more transparent to the user. Design and implementation of the functions is described in the following subsections.

6.3.1 Creating configuration files and directories

The file structure of Pog packages is different from the default described in the application documentation. To ease discovery of user-editable configuration files, Pog manifests should create the relevant configuration files and directories, even if empty. Another motivation for creating files is for applications which require a configuration file to set data paths or disable automatic updates.

Pog provides two commands for this use case – `New-Directory`, which ensures that a provided path exists and is a directory, with an optional script argument to populate the directory if it did not previously exist, and `New-File` (Listing 3), with similar functionality for creating files. `New-File` has separate script arguments generating the initial content of the file, and for updating existing content.

```powershell
# ensure auto-update is disabled
New-File "./config/settings.json" {
  ConvertTo-Json @{"update.mode" = "none"} # default content
} {
  # VS Code JSON parser is quite liberal and accepts trailing commas
  #  and comments, cannot use ConvertFrom-Json to reliably parse the file
  $old = cat -Raw $_
  $new = $old -replace '("update.mode"\s*:\s*)"([a-z][A-Z]+)"', '$1"none"'
  if ($new -ne $old) {
    Set-Content $_ $new -NoNewline
  }
}
```

Listing 3: Example of using `New-File`

Currently, there is no built-in facility for parsing common file formats, and each manifest must either use the PowerShell parsers for XML and JSON, or operate on raw strings, typically using regular expressions, which is fundamentally fragile.

In the future, Pog should provide parsers and serializers for most common configuration formats with a unified manipulation interface in `New-File`. There are multiple motivating factors: 1) There are only a few commonly used configuration formats. 2) PowerShell does not provide a built-in parser for most common formats. For some formats, especially JSON, configuration files use various language extensions (JSON comments, trailing commas) not supported by the built-in parser. 3) For configuration files, it is vital to preserve formatting and comments when modifying the file. Popular parsers for most formats only return parsed data, where this information is lost.

6.3.2 Linking files and directories

As described in Section 4.3, some applications use a fixed portable data path inside the application directory. Since Pog packages have a fixed directory structure, and overwrite the application directory on update, the data paths must be moved. This can be achieved either using symbolic links [27], or junctions and hard links [28].

The main advantage of symbolic links is the support for relative paths, which are resolved relative to the location of the symbolic link. This is convenient for portable applications, since the links do not need to be updated when the directory changes. However, creating symbolic links on Windows is a privileged operation – by default, users cannot create symbolic links, unless the user is an administrator, Developer Mode is enabled system-wide [29], [30] or an appropriate group policy is configured [31], granting the `SeCreateSymbolicLinkPrivilege` privilege to the user.

Hard links and junctions do not suffer from privilege limitations; however, they do not support relative paths. Additionally, hard links share permissions between the target and the hard link, which makes it more difficult to prevent changes to the application directory using access control while still allowing changes to a linked data file.

To move a data file or directory from the application directory, Pog provides the `New-Symlink` command. The command ensures that the target exists, either by creating an empty file/directory or by copying an existing entry from the source path, and then creates a symbolic link from the source path to the destination.

Unless the privilege requirements for symbolic links are relaxed in a future Windows version, Pog will eventually switch to using junctions and hard links, which avoid the privilege issues at the cost of increased complexity of the implementation.

6.3.3 Executable metadata configuration

Each Windows executable file, using the Portable Executable (PE) format, may have associated metadata, represented as a so-called manifest, an XML file placed either as a separate file next to the executable, or embedded inside the executable. The manifest is, among other options, used to specify that an executable requires elevation to run, describe how shared libraries are loaded and declare support for display scaling. The rest of this section focuses on controlling display scaling, since most of the other options are not currently relevant for Pog.

Windows applications may declare various levels of support for display scaling. If the application declares full support for internal scaling, the Windows graphical system does not perform any further scaling. For older applications, which do not declare support for internal scaling, the graphical system uses bitmap scaling when compositing the application, which results in blurry text for most scaling ratios.

In the author's view, the default bitmap scaling is harder to use than an unscaled application window for commonly used scaling ratios. Thus, Pog provides a command to disable display scaling for an application called `Disable-DisplayScalling`, which internally alters the executable manifest to prevent bitmap scaling. If the user prefers the default scaling behavior, he may use the built-in per-application scaling options to override the change.

6.3.4 Exposing commands and shortcuts

Most available packages expose one or more entry points to the package. There are two common types of entry points – shell links [32], more commonly known as shortcuts, typically used for graphical applications, and commands, typically used for console applications. Users should not launch executables from the application directory directly and always use the exposed entry points.

There are two main reasons for using explicit entry points. First, the application may need to be configured to run in portable mode, as described in Section 4, which is done by the entry points. Second, the public interface of the application is made explicit, and all packages have a consistent interface, easing both usage and discovery for the user and internal package management for the implementation.

To expose entry points, Pog provides a pair of commands: `Export-Shortcut` and `Export-Command`. Both commands provide a similar set of parameters, which allow the author to configure the entry point name, target path, working directory, list of additional arguments to pass to the target and a table of additional environment variables to set for the target. The author specifies relative paths inside the package, which are automatically resolved while creating the entry point.

```
Export-Shortcut "Alacritty" "./app/Alacritty.exe" `
  -Arguments @("--config-file", "./config/alacritty.yml") `
  -VcRedist

Export-Shortcut "QtRVSim" "./app/qtrvsim_gui.exe" -Environment @{
  QTRVSIM_CONFIG_FILE = "./data/qtrvsim_gui.ini"
}

Export-Command "restic" "./app/restic.exe" `
  -Arguments @("--cache-dir", "./cache")
```

Listing 4: Example of exporting entry points

Exposed shortcuts are stored as shortcut files in the root of the package directory, exposed commands are stored as custom executable launchers in an internal subdirectory inside the package. Due to the constraints described in Section 4, even shortcuts internally use launcher executables. The design and implementation of the launcher executable is detailed in Section 8.

A standing issue with the current implementation is taskbar pinning – if a user pins a running application to the taskbar, the taskbar icon references the application executable directly, bypassing the launcher. The author is not aware of any solutions to the issue without cooperation from the invoked application process, and to the author's knowledge, no other launcher implementation handles this scenario correctly. It may be possible to work around the issue by injecting a dynamic library into the application process to configure the `AppUserModelID` [33] properties of created graphical windows; this approach is currently out of scope for Pog.

6.3.5 MSVC runtime libraries

C++ applications compiled using the MSVC toolchain link either statically or dynamically to the MSVC runtime library. If the runtime is dynamically linked, the application depends on a set of shared libraries, most commonly one of the versioned `vcruntime*.dll` libraries. An application may either distribute a private copy of the libraries, or install a system-wide version using the Visual C++ Redistributable installer package [34], provided by Microsoft.

Since the runtime libraries are a very common dependency, Pog bundles an up-to-date version of the libraries. An entry point may request that the libraries are added to `PATH` using the `-VcRedist` switch parameter of the two commands above.

6.4 Exporting a package

In the previous steps, all changes were constrained to the package directory. However, users expect the installed applications to be available user-wide, the same way as standard system-wide applications. For graphical applications, this typically entails adding a shortcut to the Start menu and relevant file associations to the registry. For console applications, public binaries should be available in a directory that is added to the `PATH` environment variable, allowing the user to invoke the binary just by specifying the name instead of a full path.

Pog internally creates a `package_bin` directory, which is added to the `PATH` environment variable during initial setup. When shortcuts are exported for the first time, Pog also creates a subdirectory called `Pog` inside the user-local Start menu directory. During export, a symbolic link is created for each of the exposed commands in the package, and all exposed shortcuts are copied to the Start menu subdirectory.

Currently, in case of conflicting exports from multiple packages, Pog uses that last exported one, overwriting the previous exports. This is convenient for quick switching between multiple separate installations of a package, while keeping the canonical entry point names. If the user wishes to use multiple installations of a package without name conflicts, the exposed entry points of a package may be renamed by editing the package manifest stored inside the package directory. However, if the package is later updated, the manifest is overwritten. To avoid this issue, Pog may in the future add an option to set a prefix or a suffix for each package, which is automatically added to the name of all exports.

6.5 Uninstallation

Well-packaged Pog packages are fully self-contained, with the exception of the exported entry points. This makes clean uninstallation trivial – all exports are removed, then the package directory is deleted. Data directories may optionally be preserved for later reinstallation.

Section 7

Package repositories

Most end users are not expected to author their own packages, and use existing package repositories instead – for this use case, a fully remote repository is more convenient. However, for packaging and development, it is preferable to use a local repository, and later contribute the changes to an upstream remote repository.

Pog supports both local and remote repositories. Both types use a two level structure – a repository contains a set of packages, each package contains a set of manifests for different versions of the package.

7.1 Local repositories

Local directories are ordinary directory trees. For most packages, manifests for all versions are very similar, differing only in version, download URL and the checksum. To ease maintenance, Pog supports two types of packages for local repositories: direct and templated.

In direct packages, each version of a package is represented by a directory containing the manifest and optional auxiliary files. During import, this directory is directly copied into the package directory.

Templated packages contain a manifest template file, where some values are replaced with a template string. For each version of the package, the repository contains a data file with values that are substituted into the template when the version is imported, as shown in

```
# aria2/.template/pog.psd1
@{
  Name = 'aria2'
  Architecture = 'x64'
  Version = '{{TEMPLATE:Version}}'

  Install = @{
    Url = "{{TEMPLATE:Url}}"
    Hash = "{{TEMPLATE:Hash}}"
  }

  Enable = { <# ... #> }
}

# aria2/1.37.0.psd1
@{
  Version = '1.37.0'
  Url = 'https://github.com/aria2/aria2/releases/...'
  Hash = '67D01530...'
}
```

Listing 5: Example of a manifest template and a corresponding substitution file

Listing 5. To create the manifest for a new version, it is enough to add a new data file. If an issue with an existing package is discovered, it is typically enough to update the manifest template.

7.2 Remote repositories

Pog remote repositories are designed to be served from a static HTTP server. Pog does not have a separate update step, where a remote repository is fully downloaded before it can be used; instead, it operates directly on the remote repository over HTTP. The root of the directory contains a JSON listing of all available packages and versions; for the main repository, the listing currently takes up ~30 kB. The listing is cached locally for a short duration to speed up successive operations in the same PowerShell session. In the future, as the number of packages grows, the version listing may be split into multiple separate endpoints. Each version of a package is packaged as a ZIP archive, which is unpacked into the package directory when importing the package.

Unlike some package managers, such as Pacman, which use signatures to verify the authenticity of the manifest and the downloaded sources, Pog implicitly trusts any configured repository and leaves authentication up to the administrator of the repository. Most repositories are expected to be hosted on GitHub or GitLab, which already provide a way to authenticate users and restrict access through pull requests, simplifying the publishing experience.

The remote repository can be generated from a local repository.he expected workflow is that a local repository is tracked in Git and a CI script is used to automatically generate an up-to-date remote repository. This is how the main Pog repository [35] is maintained.

7.3 Manifest generators

Keeping track of new software releases is time consuming for a package maintainer maintaining multiple packages. To simplify the process, Pog provides support for manifest generators – a set of scripts which check for new releases and generate package manifests for new versions. Manifest generators are used with templated packages, since they only need to generate the substituted data file.

Each generator consists of two scripts: `ListVersions` and `Generate`. During an update, all versions of the packaged application are enumerated and returned by the `ListVersions` script. Pog then selects versions which do not already have a manifest and calls the `Generate` script, which typically retrieves the checksum for the application archive and returns a hash table with keys matching the template strings in the manifest template. The hash table is automatically serialized into a data file for the given version. A small set of commands is provided for retrieving checksums for remote files, working with GitHub releases and parsing checksum files; in the future, commands for other common platforms may be added.

```
@{
  ListVersions = {
    Get-GitHubRelease sharkdp/hyperfine | % {
      $Asset = $_.assets | ? name -like "hyperfine-v*-..."
      if ($Asset) {
        return @{
          Version = $_.tag_name.Substring(1)
          Url = $Asset.browser_download_url
        }
      }
    }
  }

  Generate = {
    return [ordered]@{
      Version = $_.Version
      Url = $_.Url
      Hash = Get-UrlHash $_.Url
    }
  }
}
```

Listing 6: Example of a manifest generator. `ListVersions` returns a list of all available versions, newly released versions are then passed to `Generate` and the returned object is serialized to a substitution data file for the version.

Section 8

Application launchers

As described in Section 4 and Section 6.3.4, entry points to a package often need to modify the command line arguments or environment variables passed to an application, which is done by a launcher. This section discusses possible approaches to the implementation and describes the launcher implementation used by Pog.

On POSIX-like systems, launchers are commonly implemented using shell scripts, due to a combination of platform features. First of all, shell scripts may be invoked using an ordinary `exec` system call, if the interpreter is declared using a shebang – invoking a shell script is transparent to the caller. Additionally, since the file extension is not used to disambiguate the file type, the filename of a shell script launcher may match the target. The `exec` system call may also be used by the launcher to replace its process with the target, ensuring proper delivery of signals and all other inter-process communication. The last relevant feature is argument passing – since arguments are passed as an array, no extra parsing tends to be done by the shell and the arguments may be forwarded to the target unchanged.

On Windows, there's no direct alternative to the features mentioned above. New processes are created using the `CreateProcess` and `ShellExecute` Win32 functions. `ShellExecute` is a high-level function, which supports executing files according to file associations. However, for executing binaries, the `CreateProcess` function is often used, which has lower overhead, but only supports a few specific file types: PE binaries (`.exe`), COM binaries (`.com`) and batch files. As a result, the launcher must be use one of the listed file types, otherwise callers who assume that the target is a binary and use `CreateProcess` would be unable to execute the process.

The second issue is lack of support for replacing the current process with a new executable image, as provided by `exec`. Instead of letting the caller interact directly with the target, the launcher must keep running and attempt to forward interactions between the caller and the target, primarily the exit code of the target and any attempts to terminate the target. Forwarding more complex interactions, such as window messages is likely infeasible; fortunately, such interactions are rare.

Last major issue lies in argument passing. As described in Section 4.2, arguments are passed as a single command line string, parsed by the callee, resulting in inconsistencies. This is especially gruesome for batch files, the closest analogue of Linux shell scripts on Windows, interpreted by `cmd.exe`. When a batch file is executed using `CreateProcess`, passing a command line of the form `<batch-file-path> <args>`, the actually invoked command line is `C:\WINDOWS\system32\cmd.exe /c <batch-file-path> <args>`, which results in `cmd.exe` attempting to parse both the path and the arguments as a batch command, causing both usability and security issues. [36] Arguments containing a pair of `%` are expanded as variables. If control characters like `|` and `&` appear in an argument without escaping, `cmd` assumes the following token is another command and attempts to execute it. For example, when `.\script.bat dir|` is passed as a command line to `CreateProcess`, `cmd.exe` shows a syntax error instead of passing the string `dir|` as an argument to the script.

Additionally, when Control-C is pressed while a batch file is running, the following message is printed: `Terminate batch job (Y/N)?` If the calling application assumes that it's invoking a native binary that exits immediately upon receiving Control-C, that can cause the application to hang or incorrectly interpret the command output. Both of these issues make batch files unsuitable as launchers, leaving binaries as the only option.

Launchers also have two alternative use cases: as a replacement for symbolic links, and as a substitute for shebang support. On POSIX-like systems, symbolic links are commonly used to provide multiple names for an executable, or redirect a well-known path to a different executable. On Windows, many binaries depend on shared libraries distributed alongside the binary. When the binary is executed through a symbolic link, the dynamic linker attempts to load libraries relative to the symbolic link instead of the target, unlike on Linux. Instead, launchers are often used to invoke the target. Similarly, since most scripts cannot be invoked without explicitly specifying the interpreter, unless somewhat intrusive modifications to file associations are made and `ShellExecute` is used by the caller, launchers may be used as an alternative, internally invoking the script with a specified interpreter.

8.1 Shim executables

There are multiple existing implementations of launchers using binary executables, typically called *shims*. A well-known family of implementations is the Scoop shim [37], and multiple compatible reimplementations using various approaches. The original shim is implemented in C#, with minimal features and considerable overhead due to the startup of the .NET runtime. The Scoop shim reads a text file with the `.shim` extension and base name matching the shim binary, which specifies a path to the target and optionally an argument string to prepend to the passed command line. The shim does not attempt to handle signals or ensure the target is terminated with the shim process. The alternative reimplementations are native binaries, with reasonable overhead and support most of the required features, but only support configuring the target and the additional arguments, and the file size of the binary is somewhat large (over 100 kB), given that the binary is copied for every exported shim.

Recently, the Bun[17] project developed a new shim, with a binary data format and a focus on low overhead and small file size. However, it lacks support for correct termination propagation, and only supports specifying the target, an interpreter and additional arguments.

Pog requires the shim to support configuring environment variables, arguments and the working directory, in addition to correct handling for termination and Control-C. To the author's knowledge, no existing shim implementation provides the required feature set; therefore, Pog uses a custom shim implementation, with a custom data format. Since com-

[17]https://bun.sh/

piling a new launcher each time a package is installed is not feasible, as Windows do not provide a C compiler, a pre-compiled binary is used and patched with the information necessary to launch the target application.

The following subsections contain a short description of Win32 PE binary properties relevant for the shim executable implementation in Pog.

8.1.1 PE resources

The PE format used for both Windows executable files (`.exe`) and shared libraries (`.dll`) supports embedding additional resources, which are either used by the program itself, such as cursor bitmaps, GUI elements and text translations, or by the shell, such as file icons, version data and the manifest. The resources can be inspected and modified even after the binary is compiled, using Win32 functions [38].

8.1.2 PE subsystem

Every PE executable specifies a target *subsystem* as part of the PE header, indicating the runtime environment it requires. Two subsystems in common use is the console subsystem and the graphical subsystem. Console executables are always launched with an associated console, either attaching to the parent console, or creating a new one. Graphical executables are not attached to a console on startup, and when launched from a console shell, the shell does not wait until the process terminates.

The subsystem of a PE binary can be changed post-compilation, but Windows do not provide any tooling in the Win32 library. However, since the PE binary format is well-documented, it's feasible to manipulate the subsystem with raw file I/O functions.

8.2 Pog shim executable implementation

The Pog shim is implemented in a subset of C++20, compiled without RTTI and exception support and linked without the standard library, resulting in a 10kB binary. The binary is further compressed using UPX[18], a packer which decompresses the binary on load, resulting in a 6kB final binary.

A custom binary format, designed for efficient parsing from memory, is used to configure the shim, stored as an `RCDATA` [39] PE resource inside the shim binary. This ensures that the shim can be freely copied, without dependencies on companion files. The shim data format is documented in the project[19]. All paths are resolved during shim initialization. Arguments are passed as a single serialized command line string. Environment variables are configured using a pre-processed template string, which allows the shim to build variable values using a combination of static strings and references to other environment variables.

After parsing, the shim builds the target command-line by taking the current command-line and inserting the extra launcher arguments between the stub name and the remaining arguments. Environment variables are configured on the current process and inherited by the target. The target process is spawned using `CreateProcess(...)` and assigned to

[18]https://upx.github.io/

[19]https://github.com/MatejKafka/Pog/blob/main/app/Pog/lib_compiled/Pog/src/Shim/ShimDataEncoder.cs

a Win32 job object [40] – this ensures that the target process is terminated with the shim. The shim waits for the child process to finish and exits, forwarding the exit code.

8.3 Shim executable initialization

The shim is initialized and updated from a C# library, which uses P/Invoke [41] to call the raw Win32 PE resource manipulation functions. Pog ships a pre-compiled shim binary. When a new shim is required, the binary is copied, and the library serializes the shim parameters into the `RCDATA` resource. When updating an existing shim, Pog first verifies that the configuration or the target binary resources changed, otherwise the shim is left unmodified. This speeds up reenabling packages, as the Win32 functions for updating the shim tend to be somewhat slow (~30 ms per updated file on the author's laptop).

Additionally, the target executable is analyzed, the PE subsystem of the shim is updated to match the target (this ensures that shim behaves the same as the target when invoked) and relevant resources (icons, version data) are copied to the shim. The resulting shim has the same subsystem, icon and version as the target.

Section 9

Evaluation

In this section, Pog is evaluated based on the design goals introduced in Section 3 and its performance is compared to the other three package managers discussed in previous sections.

9.1 Encapsulation and portability

The first design goal, *Encapsulation and portability*, has been satisfied for most packages by storing all application data inside the package directory and separating the installation and configuration of a package. The separation allows the user to install an application once and reconfigure it every time it is moved to a new machine, as shown in Listing 7. While this presents more friction for the user than automatically configuring the package on each invocation, it significantly speeds up the application startup.

Some compromises in encapsulation had to be made to achieve compatibility with commonly used software. The most problematic group are applications with internal support for installing packages, such as IDEs and language package managers. Unless the application provides an interface for querying data paths and enforces its use, the internally

```
PS E:\> irm "https://pog.matejkafka.com/install.ps1" | iex  # install Pog
...
PS E:\> pog tealdeer  # install tealdeer
PS E:\> tldr --show-paths
Config dir:      E:\tealdeer\config\ (env variable)
Config path:     E:\tealdeer\config\config.toml
...
```

```
PS F:\> .\Pog\setup.ps1  # setup Pog for the new system
...
PS F:\> # enable and export all installed packages
     >> Get-PogPackage | Enable-Pog -PassThru | Export-Pog
WARNING: Overwriting existing command '7z'...
WARNING: Overwriting existing shortcut 'OpenedFilesView'...
WARNING: Overwriting existing command 'OpenedFilesView'...
WARNING: Overwriting existing command 'tealdeer'...
WARNING: Overwriting existing command 'tldr'...
PS F:\> tldr --show-paths
Config dir:      F:\tealdeer\config\ (env variable)
Config path:     F:\tealdeer\config\config.toml
...
```

Listing 7: Example of moving an installed package between machines. The package used, `tealdeer`, supports listing the data paths used by passing the `--show-paths` flag. In the first listing, Pog is installed on a USB flash drive and used to install `tealdear`. Afterwards, the flash drive is moved to another computer, Pog is reactivated and then used to reconfigure `tealdear` to run on the new system, shown in the second listing.

installed packages tend to use non-portable data paths. Pog cannot effectively redirect these paths without some form of sandboxing.

A related group are badly-behaved applications, where data important to end users can be stored portably, but other ephemeral data, such as crash logs or caches, are stored user-wide. Many vendors implement portable mode due to requests from the community but do not actively use it, resulting in common issues in various edge cases.

Finally, many applications currently do not support any form of portable mode and, therefore, cannot be packaged for Pog. This is especially common for games and applications that integrate deeply with the system. Fortunately, for developer tooling, the area most relevant for the author and the target audience, some form of support for portable mode is usual.

9.2 Minimal external dependencies

The second design goal, *Minimal external dependencies*, is also satisfied. Existing packages do not have any external dependencies, with the exception of MSVC runtime libraries, which are a such a frequent dependency that Pog bundles the latest version and makes it available to any package that requests it. For applications which do not bundle all dependencies in their binary releases, multiple source archives may be requested in the package manifest. Multiple versions of a package may be installed side-by-side using different target names, as shown in Listing 8.

9.3 Transparent installation & Accessibility

The goal of *transparent installation* is achieved by avoiding the use of closed-source installers, and by using PowerShell, a well-known scripting language in the Windows environment, as the public interface and the package manifest format for Pog. Since the package manifest is placed into the package directory when imported, the user is free to modify the installation and configuration process as they see fit.

```
PS D:\> pog Node.js 21.0.0 -TargetName node-21
PS D:\> pog Node.js 22.0.0 -TargetName node-22
WARNING: Overwriting existing command 'node'...
WARNING: Overwriting existing command 'npm'...
WARNING: Overwriting existing command 'npx'...
PS D:\> node -v
v22.0.0
PS D:\> Export-Pog node-21
WARNING: Overwriting existing command 'node'...
WARNING: Overwriting existing command 'npm'...
WARNING: Overwriting existing command 'npx'...
PS D:\> node -v
v21.0.0
PS D:\>
```

Listing 8: Example of installing multiple versions of a package using different target names and switching between them.

The terminology and model of installation is deliberately kept simple, with the minimal level of abstraction necessary to ensure a consistent user experience. By utilizing the PowerShell platform, Pog provides easily scriptable native PowerShell commands, including completions for all common parameters and a .NET object-oriented API for more advanced scripting needs. For a practical demonstration of the interface, see Appendix A.

9.4 Packaging experience

The use of the PowerShell data file format ensures that Pog package manifests are likely familiar to many Windows developers. Unlike the more common formats used by other package managers, such as JSON and YAML, Pog manifests support comments, compact whitespace-based notation for arrays and tables, and inline scripts with correct syntax highlighting in many popular text editors. For comparison, manifests for the Go programming language package for Pog, Scoop and WinGet are shown in Appendix B, Appendix C and Appendix D, respectively.

Since Pog packages use binary releases, new packages may be authored by anyone, without requiring access to the original source code. No repackaging is necessary, as Pog provides a library of functions to adapt software to work well as a portable package during package installation. To keep packages up-to-date, maintainers can use manifest generators, which automatically generate manifests for newly released versions of the upstream package without any manual work. To avoid duplication, a manifest may be specified using a template and corresponding substitution files for each version.

At least for the author, Pog has also proven to be a useful tool for local development. When building a project from source, the user can create a new package directory inside a registered package root, build the project inside the `./app` directory and then provide a package manifest to integrate it with Pog. Since Pog does not keep a separate database of installed packages, locally-added packages behave exactly the same as packages installed from the repository.

9.5 Performance evaluation

While performance was not explicitly stated as one of the design goals, it is highly relevant for practical use. Since disk and network usage tend to be similar for all package managers and memory usage is not a limiting factor, the primary metric for most users is likely the speed of a few common operations – installation of a new package, update of an existing package, searching for packages in a repository and listing installed packages.

I do not see a fair way to compare the search and listing speed, since Pog has an order of magnitude fewer packages than the better established alternatives. However, since Pog can operate directly on the remote repository, and caches the package listing in memory between operations, I do believe it should be competitive, especially as it avoids the startup penalty by using the already-loaded PowerShell runtime. This seems to be confirmed by my limited manual testing.

9.5.1 Package installation speed

There are two relevant metrics for installation speed – the time taken to install a single package and to install multiple packages. However, none of the four tested package man-

Package	Version	Archive size	Installer size	Description
Alacritty	0.12.3	5 MB	2.2 MB (MSI)	A simple terminal emulator
go-lang	1.22.3	73 MB	60 MB (MSI)	Go programming language
Firefox	126.0	N/A	62 MB (7zip*)	Mozilla Firefox
CMake	3.29.3	43.6	32.5 MB (MSI)	CMake build system
Sublime Text	4169	22.1 MB	16 MB (Inno)	Text editor
ccache*	4.9.1	1.57 MB	N/A	Compiler cache
Typst*	0.11.1	13.9 MB	N/A	Typst typesetting system

Table 2: A table of all tested packages, specifying the versions and download sizes. The Firefox installer is a self-extracting 7zip archive. For the installation test, the first six packages were used. For the remaining tests, `ccache` was replaced by `Typst`.

agers currently support parallel installation of multiple packages; therefore, it is sufficient to focus on the speed of installing a single package.

Since the relative installation speed may depend on the size of the downloaded files, the number of files accessed during the installation and the complexity of the installer or the corresponding installation script, multiple different packages were tested, noted in Table 2, including the size of the downloaded archive, used by Scoop and Pog, and the installer, used by Chocolatey and WinGet. The tested package managers were Pog v0.8.1, Scoop v0.4.2, Chocolatey v2.2.2 and WinGet v1.7.11261.

Each package manager was installed into a clean Windows Sandbox instance, running on Windows 10 2004, on a laptop with Intel i5-8350u, 16 GB of RAM, a recent NVMe drive and a 300 Mbps network connection. All tests were executed from a clean PowerShell instance. To warm up each package manager and update the package repositories, two packages (7zip and Git) were installed. Additionally, MSVC runtime libraries required by some of the packages were installed. For Scoop, the `extras` repository containing graphical applications was added.

Following the warmup, the tested packages were downloaded in random order, measuring the total time from the invocation of the installation command to its completion. The whole process was repeated 32 times for each package manager, each time creating a new Windows Sandbox instance. Since the measurements were done on a desktop Windows system with many active background services, significant outliers (2–3 for each package manager) were removed.

Figure 1 shows the total time taken to install all packages in each iteration. Both Scoop and Pog, installing from archives, are significantly faster than both installer-based package managers, confirming the hypothesis that installation speed can be significantly improved by avoiding installers.

The total installation time is strongly dependent on the performance of installing the larger packages; to illustrate the differences depending on package size, Figure 2 shows mean installation times separately for each package. Both Scoop and Pog are consistently

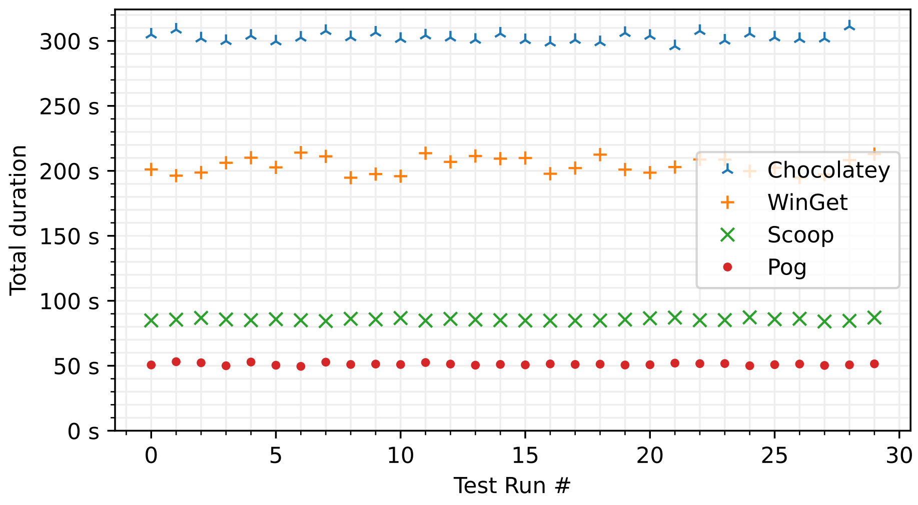

Figure 1: Time taken to install the first 6 packages from Table 2. Packages were installed in random order, each package manager was warmed up by installing two medium-sized packages. Since the tests were ran on a desktop system, any significant outliers (2-3 for each package managers) were removed.

faster than the remaining two, with Scoop being slightly faster for smaller packages and Pog taking the lead for larger packages.

The significant discrepancy in installation time for Alacritty is likely caused by the Pog package requesting both the application binary and a default configuration file, while Scoop only downloads the binary.

Another notable difference is between the `go-lang` and `Firefox` packages, which can be explained by the number of unpacked files – although the archive size is similar, the ex-

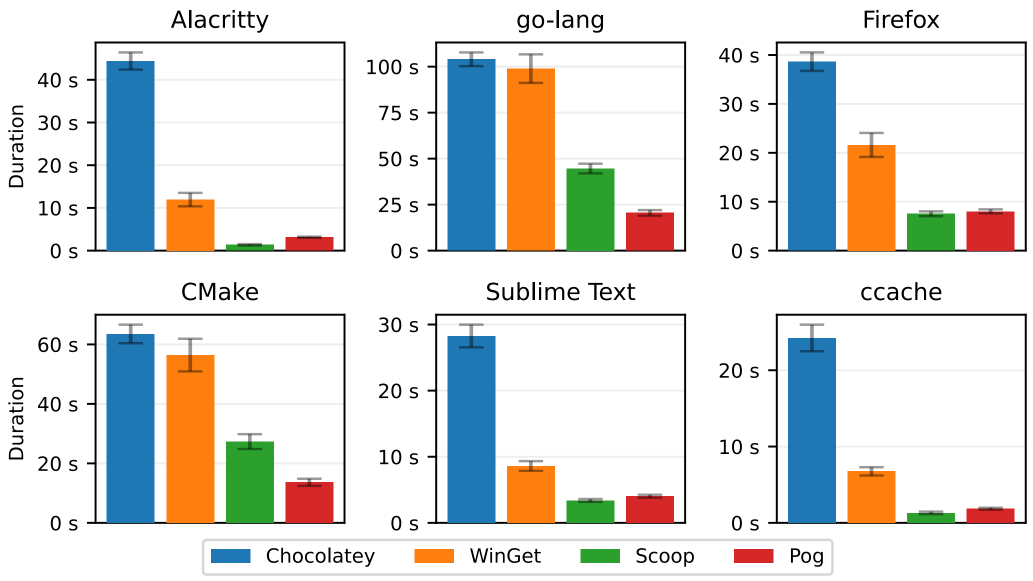

Figure 2: Time taken to install each of the first 6 packages from Table 2 separately. The plot shows the arithmetic mean and standard deviation from all iterations.

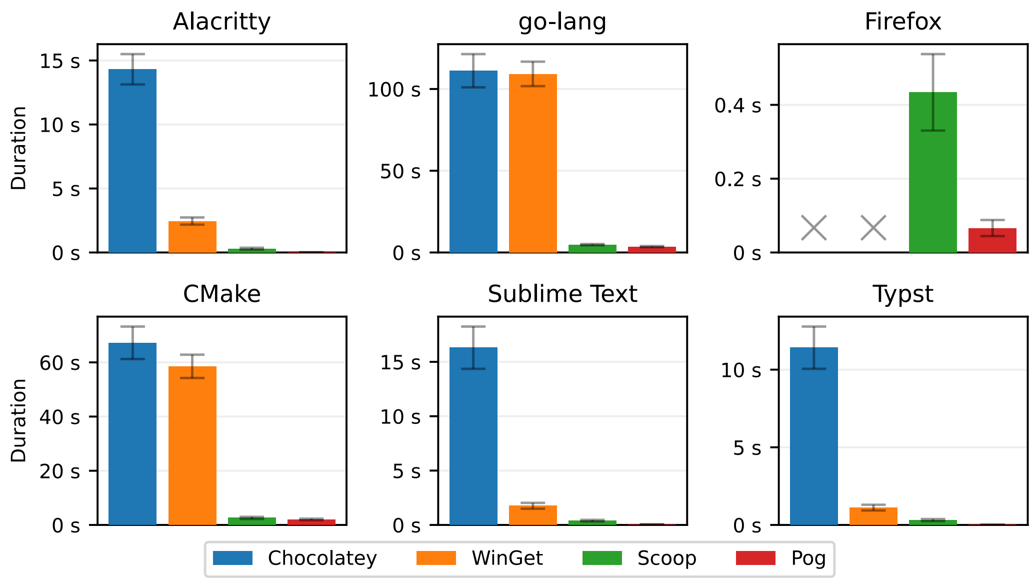

Figure 3: Time taken to uninstall each of the tested packages. The plot shows the arithmetic mean and standard deviation from all iterations. Crosses indicate invalid measurements due to technical issues.

tracted `Firefox` package contains 61 files, while the `go-lang` package has 12882 files. Since both Scoop and Pog install the packages from the same archives, the difference indicates that Pog is faster at handling archives with many files.

9.5.2 Package uninstallation speed

To measure the speed of uninstalling a package, a similar setup was used, installing fixed versions of all tested packages. Following, the packages were uninstalled one-by-one, measuring the time until the invoked command completed. Instead of `ccache`, this test and the update test below use `Typst`, as WinGet does not provide older versions of `ccache`, which were needed to test update performance.

The mean uninstallation times for each package are shown in Figure 3; for Firefox, the measurements for WinGet and Chocolatey are not included since WinGet cannot uninstall it automatically and just shows the graphical installer dialog, while Chocolatey prompts for interactive confirmation before uninstalling the package that the author could not suppress. The figure shows an order-of-magnitude difference in performance between the installer-based and archive-based package managers.

9.5.3 Package update speed

To measure the speed of updating a package, an older version of all tested packages was installed. Then, the packages were updated to a fixed newer version one-by-one, while measuring the time taken until the invoked command completed. The mean time for each application is again shown in Figure 4. Again, both archive-based package managers are significantly faster, primarily due to faster uninstallation of the previous package version. Notably, none of the installers used by WinGet and Chocolatey seem to use differential updates, where only the changed files are updated, as the update time matches the sum of installation and uninstallation times.

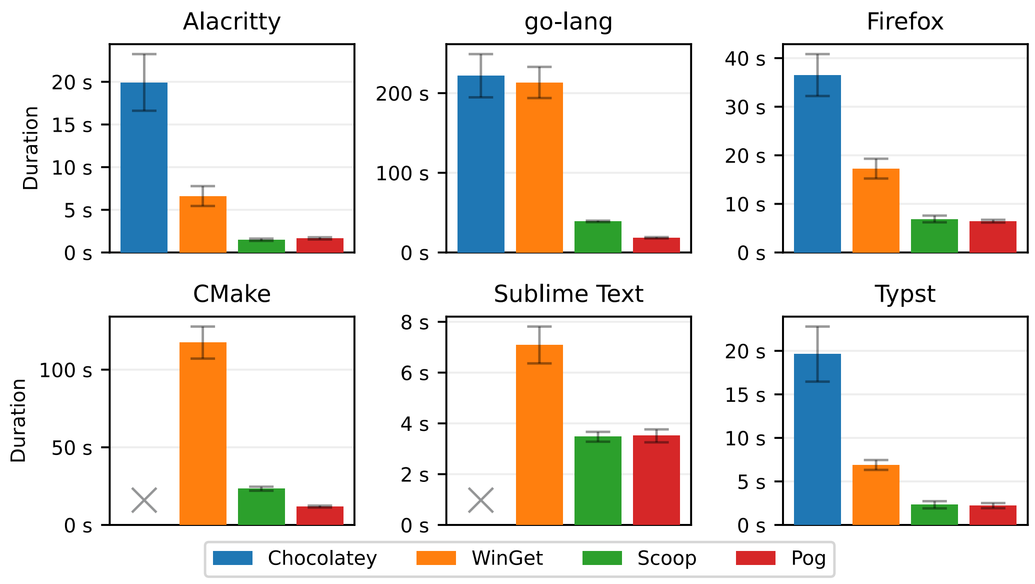

Figure 4: Time taken to update each of the tested packages. The plot shows the arithmetic mean and standard deviation from all iterations. Crosses indicate invalid measurements due to technical issues.

Chocolatey does not have the required older versions of CMake and Sublime Text in its repositories; selecting package versions available from all tested package managers proved to be surprisingly problematic. Therefore, the measurements for these two packages are missing.

Another issue affecting the results is that Scoop does not seem to support updating a package to a specific version, only to the latest one. To avoid this issue, the old package version was first uninstalled and then the updated version was installed instead of updating directly. This matches what Scoop does during an update, but there might be a slight overhead from the two separate command invocations and the unnecessary setup of the application directory.

Section 10

Future work

After almost four years of development, Pog is nearing a public release. In its current state, it is entirely usable for most of its stated use cases, and the author has been using it as the primary package manager on their Windows system for the past two years.

However, there are still a few important missing pieces. First, the installation process may leave the package in a somewhat inconsistent state if the installation fails. While each of the four installation phases leaves the relevant parts of the package in a valid state, the package as a whole may not be consistent.

As an example, when a package is updated to a new version, if the import of the new manifest succeeds, but then the installation fails while downloading an archive, the package will have the new manifest, but the actual installed version of the application will be left unchanged. Similarly, if the configuration script fails, the package may misbehave when invoked through an entry point.

Most inconsistent states can be resolved by re-running the installation command after fixing the original issue. However, for an uninitiated user, the current behavior may be confusing. A perfect solution is likely infeasible while allowing arbitrary configuration scripts. However, for other phases of the installation process, technologies such as Transactional NTFS may be used to roll back changes on failure. As an additional measure, the exported entry points may be hidden while operating on the package to prevent the user from using the package. In case of an error, the user will be forced to get the package back to a consistent state before using it again, preventing them from using a potentially misbehaving package.

Another missing piece is support for the ARM architecture. With the major performance improvements in the last few years, ARM-based computers are becoming increasingly popular. Currently, Pog and all published packages only support x64; while adding ARM support to Pog itself should be easy, architecting support for packages of different architectures while moving the installation between machines may prove tricky.

As already discussed, many applications do not support portable mode. Since a user will likely need some of these applications, they must currently use another package manager to acquire them. One option for Pog would be to provide a separate repository of packages still managed by Pog but using the default non-portable data paths. The effort to implement this should be fairly minimal, while still providing many of the benefits of Pog.

Non-portable applications could also be supported using some form of user-space sandboxing that would transparently redirect access to specific paths, as described in Section 4.4. It should be possible to extend the current executable shim to inject a dynamic library to the target process and implement the sandbox by hijacking relevant Windows API calls.

Finally, what is still missing are the packages. The main Pog repository currently contains 174 packages, an order of magnitude fewer than Scoop and other alternatives. While Pog is unlikely to ever have a similar number of packages due to its hard requirement of portability, there is still a lot of room for growth, both by adding new packages and convincing vendors and especially open-source projects to implement support for portable mode.

Section 11

Conclusion

In this thesis, I presented Pog[20], a portable package manager for Windows. The project proves the feasibility of building a package manager that fully encapsulates applications and shows that useful features and significant performance improvements are gained in doing so.

I believe this thesis presents the most comprehensive overview of package management on Windows available today, analyzing both inherent and avoidable issues with packaging on the platform, discussing effective solutions for most and contrasting the approaches taken by existing alternatives.

Pog is not purely a research project, but also a practical, robust, open-source tool, intended to simplify application management for power users from around the world. After a long development, Pog is nearing an official public release. Hopefully, it finds its place in the Windows community.

[20]https://pog.matejkafka.com/

Bibliography

[1] E. Dolstra, M. de Jonge, and E. Visser, “Nix: A Safe and Policy-Free System for Software Deployment,” in *Proceedings of the 18th USENIX Conference on System Administration*, in LISA '04. Atlanta, GA: USENIX Association, 2004, pp. 79–92.

[2] NixOS Foundation et al., “Nix & NixOS | Declarative builds and deployments.” [Online]. Available: https://nixos.org/

[3] The kernel development community, “Squashfs 4.0 Filesystem — The Linux Kernel documentation.” [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/squashfs.html

[4] Microsoft, “Windows Installer - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/msi/windows-installer-portal

[5] Microsoft, “Standard Actions - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/msi/standard-actions

[6] Microsoft, “About Transforms - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/msi/about-transforms

[7] S. Asmul, “The corporate benefits of using MSI files - Server Fault.” [Online]. Available: https://serverfault.com/questions/11670/the-corporate-benefits-of-using-msi-files/274609#274609

[8] Microsoft, “Windows Installer - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/msi/windows-installer-portal

[9] Microsoft, “Universal Windows Platform documentation.” [Online]. Available: https://learn.microsoft.com/en-us/windows/uwp/

[10] Chocolatey Software, Inc., “Chocolatey - The Package Manager for Windows.” [Online]. Available: https://chocolatey.org/

[11] “Chocolatey Software | powershell-core (Install) 7.5.0-preview02.” [Online]. Available: https://community.chocolatey.org/packages/powershell-core#files

[12] “NuGet does not deal with blocking version conflicts from existing installed packages · Issue #116 · chocolatey/choco · GitHub.” [Online]. Available: https://github.com/chocolatey/choco/issues/116

[13] Microsoft, “Microsoft/Winget-CLI: Windows Package manager CLI (aka Winget).” [Online]. Available: https://github.com/microsoft/winget-cli

[14] K. Beigi, “The Day AppGet Died..” [Online]. Available: https://keivan.io/the-day-appget-died/

[15] “Scoop.” [Online]. Available: https://scoop.sh/

[16] “Persistent data · ScoopInstaller/Scoop Wiki · GitHub.” [Online]. Available: https://github.com/ScoopInstaller/Scoop/wiki/Persistent-data

[17] “Dependencies · ScoopInstaller/Scoop Wiki · GitHub.” [Online]. Available: https://github.com/ScoopInstaller/Scoop/wiki/Dependencies

[18] “Why PowerShell · ScoopInstaller/Scoop Wiki · GitHub.” [Online]. Available: https://github.com/ScoopInstaller/scoop/wiki/Why-PowerShell

[19] Microsoft, “Understanding how packaged desktop apps run on Windows - MSIX | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes

[20] A. Ustynov, “Light-Weight Sandbox for Installers,” 2022. [Online]. Available: https://dspace.cvut.cz/handle/10467/101140

[21] “Isolation Mechanism | Sandboxie Documentation.” [Online]. Available: https://sandboxie-plus.github.io/sandboxie-docs/Content/IsolationMechanism.html

[22] Microsoft, “about Data Files - PowerShell.” [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_data_files?view=powershell-7.3

[23] Microsoft, “Background Intelligent Transfer Service - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/bits/background-intelligent-transfer-service-portal

[24] “JSON Lines.” [Online]. Available: https://jsonlines.org/

[25] Microsoft, “SetFileInformationByHandle function (fileapi.h) - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-setfileinformationbyhandle

[26] Microsoft, “Creating Runspaces - PowerShell | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/developer/hosting/creating-runspaces

[27] Microsoft, “Creating Symbolic Links - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/fileio/creating-symbolic-links

[28] Microsoft, “Hard links and junctions - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/fileio/hard-links-and-junctions

[29] Microsoft, “CreateSymbolicLinkW function (winbase.h) - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsymboliclinkw

[30] “Symlinks in Windows 10! - Windows Developer Blog.” [Online]. Available: https://blogs.windows.com/windowsdeveloper/2016/12/02/symlinks-windows-10/

[31] Microsoft, “Create symbolic links - Windows 10 | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-10/security/threat-protection/security-policy-settings/create-symbolic-links

[32] Microsoft, “Shell Links - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/shell/links

[33] Microsoft, “Application User Model IDs (AppUserModelIDs) - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/shell/appids

[34] Microsoft, “Latest supported Visual C++ Redistributable downloads | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist

[35] “GitHub - MatejKafka/PogPackages: Package manifests for the Pog package manager (https://github.com/MatejKafka/Pog)..” [Online]. Available: https://github.com/MatejKafka/PogPackages

[36] “BatBadBut: You can't securely execute commands on Windows - Flatt Security Research.” [Online]. Available: https://flatt.tech/research/posts/batbadbut-you-cant-securely-execute-commands-on-windows/

[37] “GitHub - ScoopInstaller/Shim: A Scoop helper program for shimming executables.” [Online]. Available: https://github.com/ScoopInstaller/Shim

[38] Microsoft, “Using Resources - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/menurc/using-resources

[39] Microsoft, “RCDATA resource - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/menurc/rcdata-resource

[40] Microsoft, “Job Objects - Win32 apps | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/procthread/job-objects

[41] Microsoft, “Platform Invoke (P/Invoke) | Microsoft Learn.” [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

Appendix A

Pog scripting

The following listings illustrate the PowerShell interface provided by Pog, focusing on examples of interactive scripting.

Completion of available versions when installing a package:

```
> pog Node.js 22.<Ctrl+Space>
22.2.0  22.1.0  22.0.0
```

Show basic information about installed packages:

```
> Get-PogPackage Alacritty, Firefox, Chromium

   Package Root: D:\_

PackageName          Version
-----------          -------
Alacritty            0.12.3
Firefox              127.0b3

   Package Root: D:\_custom

PackageName          Version
-----------          -------
Chromium             116.0.5845.97
```

List shortcuts exposed by a specific package:

```
> (Get-PogPackage Firefox).ExportedShortcuts

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          2023-09-05    16:33       1140 Firefox Private Browsing.lnk
-a---          2023-09-05    16:33       1001 Firefox.lnk
```

Find a specific package version in the repository and list the source URLs:

```
> (Find-PogPackage Alacritty 0.11.0).Manifest.Install.SourceUrl
https://github.com/alacritty/.../v0.11.0/Alacritty-v0.11.0-portable.exe
https://github.com/alacritty/.../v0.11.0/alacritty.yml
```

Manually list outdated packages:

```
> foreach ($p in Get-PogPackage) {
>   # find the latest version of the corresponding repository package
>   $r = try {Find-PogPackage $p.ManifestName} catch {continue}
>   if ($r.Version -gt $p.Version) {
>     $p | select PackageName, Version, {$r.Version}
>   }
> }
PackageName Version $r.Version
----------- ------- ----------
cURL        8.7.1_9 8.8.0_1
Firefox     127.0b3 127.0b5
Syncthing   1.27.7  1.27.8-rc.1
Thunderbird 127.0b1 127.0b2
```

Appendix B

Pog manifest example

Source:
https://github.com/MatejKafka/PogPackages/blob/main/go-lang/.template/pog.psd1

```
@{
    Name = "go-lang"
    Architecture = "x64"
    Version = '1.22.3'

    Install = @{
        Url = {"https://go.dev/dl/go$($this.Version).windows-amd64.zip"}
        Hash =
'CAB2AF6951A6E2115824263F6DF13FF069C47270F5788714FA1D776F7F60CB39'
    }

    Enable = {
        New-Directory "./config"
        New-Directory "./cache/build-cache"
        New-Directory "./cache/mod-cache"
        New-Directory "./data/go-bin"
        New-Directory "./data/packages"

        Export-Command "go" "./app/bin/go.exe" -Environment @{
            GOROOT = "./app"
            GOBIN = "./data/go-bin"
            GOENV = "./config/goenv"
            GOCACHE = "./cache/build-cache"
            GOMODCACHE = "./cache/mod-cache"
            GOPATH = "%GOPATH%", "./data/packages"
        }
    }
}
```

Appendix C

Scoop manifest example

Source: https://github.com/ScoopInstaller/Main/blob/master/bucket/go.json

Note that the Scoop manifest also includes metadata necessary to automatically update it when a new version is released. Both Pog and WinGet use a separate mechanism for manifest updates.

```
{
    "version": "1.22.3",
    "description": "An open source programming language that makes it easy
to build simple, reliable, and efficient software.",
    "homepage": "https://golang.org",
    "license": "BSD-3-Clause",
    "architecture": {
        "64bit": {
            "url": "https://dl.google.com/go/go1.22.3.windows-amd64.zip",
            "hash":
"cab2af6951a6e2115824263f6df13ff069c47270f5788714fa1d776f7f60cb39"
        },
        "32bit": {
            "url": "https://dl.google.com/go/go1.22.3.windows-386.zip",
            "hash":
"f60f63b8a0885e0d924f39fd284aee5438fe87d8c3d8545a312adf43e0d9edac"
        },
        "arm64": {
            "url": "https://dl.google.com/go/go1.22.3.windows-arm64.zip",
            "hash":
"59b76ee22b9b1c3afbf7f50e3cb4edb954d6c0d25e5e029ab5483a6804d61e71"
        }
    },
    "extract_dir": "go",
    "installer": {
        "script": [
            "$envgopath = \"$env:USERPROFILE\\go\"",
            "if ($env:GOPATH) { $envgopath = $env:GOPATH }",
            "info \"Adding '$envgopath\\bin' to PATH...\"",
            "Add-Path -Path \"$envgopath\\bin\" -Global:$global -Force"
        ]
    },
    "uninstaller": {
        "script": [
            "$envgopath = \"$env:USERPROFILE\\go\"",
            "if ($env:GOPATH) { $envgopath = $env:GOPATH }",
            "info \"Removing '$envgopath\\bin' from PATH...\"",
            "Remove-Path -Path \"$envgopath\\bin\" -Global:$global"
        ]
    },
    "bin": [
        "bin\\go.exe",
        "bin\\gofmt.exe"
    ],
    "checkver": {
        "url": "https://golang.org/dl/",
        "regex": "go([\\d.]+)\\.windows-"
    },
    "autoupdate": {
        "architecture": {
            "64bit": {
                "url": "https://dl.google.com/go/go$version.windows-amd64.
zip"
            },
            "32bit": {
                "url": "https://dl.google.com/go/go$version.windows-386.
zip"
            },
            "arm64": {
                "url": "https://dl.google.com/go/go$version.windows-arm64.
zip"
            }
        },
        "hash": {
            "url": "$url.sha256"
        }
    }
}
```

Appendix D

WinGet manifest example

Source: https://github.com/microsoft/winget-pkgs/blob/master/manifests/g/GoLang/Go/1.22.3/GoLang.Go.yaml

```yaml
PackageIdentifier: GoLang.Go
PackageVersion: 1.22.3
DefaultLocale: en-US
ManifestType: version
ManifestVersion: 1.6.0
```

Source: https://github.com/microsoft/winget-pkgs/blob/master/manifests/g/GoLang/Go/1.22.3/GoLang.Go.installer.yaml

```yaml
PackageIdentifier: GoLang.Go
PackageVersion: 1.22.3
InstallerLocale: en-US
InstallerType: wix
Scope: machine
InstallModes:
- interactive
- silent
- silentWithProgress
UpgradeBehavior: install
Commands:
- go
FileExtensions:
- go
- gohtml
ReleaseDate: 2024-05-07
Installers:
- Architecture: x86
  InstallerUrl: https://go.dev/dl/go1.22.3.windows-386.msi
  InstallerSha256:
589FA5DE9F7A2AFFFB4D98535E7DA5E1B21E8367ABBAE35168CA52FD71DB6A4F
  ProductCode: '{AF6F03B4-000D-4131-8A13-88F4C0106CA8}'
  AppsAndFeaturesEntries:
  - DisplayName: Go Programming Language 386 go1.22.3
    UpgradeCode: '{1C3114EA-08C3-11E1-9095-7FCA4824019B}'
- Architecture: x64
  InstallerUrl: https://go.dev/dl/go1.22.3.windows-amd64.msi
  InstallerSha256:
11F1A4A65C90088E942CEF4CBF286E07FD5E04F0A3E677646459415DAAFA0F4B
  ProductCode: '{3F816537-9FDB-4FE7-86E3-BBEEDFD4038E}'
  AppsAndFeaturesEntries:
  - DisplayName: Go Programming Language amd64 go1.22.3
    UpgradeCode: '{22EA7650-4AC6-4001-BF29-F4B8775DB1C0}'
```

```yaml
- Architecture: arm64
  InstallerUrl: https://go.dev/dl/go1.22.3.windows-arm64.msi
  InstallerSha256:
DC65B6E08281A50791BB86898CE96DB299BF3D27967A32F3EA1F8078E2C37B94
  ProductCode: '{C265D735-7314-4CD2-A7AC-C96DF4F9FCED}'
  AppsAndFeaturesEntries:
  - DisplayName: Go Programming Language arm64 go1.22.3
    UpgradeCode: '{21ADE9A3-3FDD-4BA6-BEA6-C85ABADC9488}'
ManifestType: installer
ManifestVersion: 1.6.0
```