RFC: Adding support for digitally signed plugins to HDF5

Glenn Song
Dana Robinson
Scot Breitenfeld

The HDF Group

Plugins are valuable extensions that enhance HDF5's functionality. They play a crucial for adding custom features, such as various compression filters and VFDs, without requiring extensive changes to the source code. To improve HDF5's security, we are exploring the option of digitally signing plugins, which would help verify and secure any plugins that a user may use.

Contents

1 Introduction

This RFC proposes a framework for integrating digital signatures into HDF5 plugins. HDF5 relies heavily on extensible *plugins*, such as compression filters, Virtual File Drivers (VFDs), and Virtual Object Layer (VOL) connectors. These plugins greatly enhance HDF5's capabilities by offering specialized functionality tailored to specific user needs.

However, the existing plugin system lacks inherent security features. This vulnerability poses potential risks, including:

- **Malicious Plugins:** Malicious actors could create and distribute tampered plugins that corrupt data, steal sensitive information, or introduce vulnerabilities into the HDF5 system.
- **Unintentional Modifications:** Even unintended modifications to a plugin, such as accidental code alterations or incorrect recompilation, can result in unexpected behavior, data corruption, or introduce security risks.
- **Supply Chain Attacks:** Compromised plugin development environments or distribution channels may result in the spread of malicious or modified plugins.

This RFC suggests integrating digital signatures into the HDF5 plugin system to mitigate these risks. Digital signatures offer a cryptographic mechanism to:

- **Verify Plugin Authenticity:** HDF5 can confirm a plugin’s authenticity by comparing its signature with a known public key, ensuring it comes from the intended source.
- **Detect Tampering:** Modifying the plugin after it has been signed will invalidate the signature, alerting users to potential tampering or corruption.
- **Enhance Trust:** Digital signatures ensure plugin integrity, enabling users to confidently utilize plugins from reliable sources without concern for malicious activity.

This proposal aims to create a secure and reliable plugin ecosystem for HDF5 while ensuring backward compatibility. Users who do not require or desire digital signature verification can continue to use HDF5 and its plugins as they currently do. However, for users who prioritize security, the

proposed framework will provide a simple and efficient way to utilize digital signatures to enhance the security and trustworthiness of their HDF5 workflows.

Key Enhancements:

- **Enhanced Security:** Reduces risks linked to harmful plugins, accidental modifications, and supply chain attacks.
- **Increased Trust:** Fosters confidence in the plugin ecosystem by offering a way to verify plugin authenticity and integrity.
- **Enhanced Data Integrity:** Safeguards data against corruption from tampered or harmful plugins.
- **Backward Compatibility:** Guarantees minimal disruption for current users who do not need digital signature verification.

The following sections detail the proposed design, implementation specifics, security considerations, and a roadmap for incorporating digital signatures into the HDF5 plugin system to achieve these key enhancements.

2 Approach: Implementing Signed Plugins for HDF5

This approach focuses on leveraging digital signatures to enhance the security of HDF5 plugins. Overview of the core mechanisms:

- **Signing:** Plugin developers use an encryption tool (in this case, GPG) to generate a key pair (private and public keys). They then sign their plugins using their private key. This creates a detached signature file, keeping the plugin and signature separate for clarity.
- **Verification:** HDF5 will be extended to allow users to verify the authenticity of a plugin. During plugin loading, HDF5 will:
    - Locate the corresponding signature file.
    - Use the plugin developer's public key to verify the signature.
    - If verification succeeds, load the plugin.
    - If verification fails, prevent the plugin from loading and notify the user.

Several other encryption tools and libraries were considered, including LibTomCrypt, LibSodium, NaCl, BouncyCastle, and OpenSSL. Ultimately, these were all set aside in favor of GPG and GPGme. If it's later determined that GPG isn’t the right tool for this purpose, the others can still be explored as alternatives. GPG was chosen for the following reasons:

- GPG is free and open source, so it can be easily modified and improved if necessary.
- GPG is widely used and works on Linux, Windows, and MacOS.
- GPG makes it easy to swap between multiple keys using its keyring feature.
- GPG supports multiple encryption algorithms such as RSA, DSA, SHA2, etc.

To provide a better understanding of GPG, it stands for GNU Privacy Guard, which is a free and open-source implementation of the OpenPGP standard. In this context, GPG has been chosen for its

features related to digital signatures, but it serves as a universal crypto engine suitable for encrypting data and communications. It can easily be used through the command line, and it can also be utilized within scripts and programs, which is important for our application.

This approach using digital signatures and GPG necessitates adjustments to various workflows. First, plugin creators must learn to generate key pairs and sign their plugins. They must also adopt and distribute their plugins and the corresponding signature files to guarantee authenticity. The strategy for plugin creators is summarized as follows:

- The plugin is assumed to be released and registered with the HDF Group, leading to an assigned registration ID by the HDF Group.
- Alternatively, if plugin developers create custom filters for specific needs that are not necessarily public, they must assign them unique IDs according to their plugin's established convention. These IDs are considered "*private*" because they are not part of the official HDF5 plugin registry and are not guaranteed to be recognized by other HDF5 installations.
- Plugin creators are required to generate and manage their key pairs. They must also sign their plugins using their private keys and GPG software to create a digital signature.
    - Users must receive both the plugin and the accompanying signature file. Therefore, creators may need to update their build processes to include these extra steps.
    - If the plugin is released from a public repository that supports workflows, for example, GitHub, the procedure could be automated as part of a release workflow action.
- Users need to access the plugin creator's public key to verify the signature. This key can be obtained from a trusted source, such as a public key server, the plugin repository, or the creator's website. More details will follow later. Essentially, signing the plugin with the private key provides digital assurance of its origin and integrity, which is crucial for the next group affected by the changes: its users.

Plugin users must obtain and install GPG/GPGME software (or a compatible alternative) on their systems. This entails learning basic GPG usage, such as importing public keys and verifying signatures. Users must also integrate signature verification into their HDF5 workflows to authenticate their plugins.

However, this approach has potential challenges. One major hurdle is user friendliness; requiring users to install and learn GPG can pose a significant barrier to adoption. Furthermore, managing and distributing public keys can be complex, particularly in larger collaborative environments where coordination between multiple users and developers is necessary. Additionally, relying solely on GPG may limit flexibility and might not be suitable for all users or environments.

A user-friendly interface that simplifies the signing and verification process is essential to address these challenges. This tool could assist users in generating and managing keys, offer an intuitive platform for signing and verifying plugins, and securely manage key storage and distribution, ultimately ensuring a smoother transition for everyone involved.

2.1 Use Cases

Several important considerations apply to typical use cases for this feature. Because security can be critical to users in many fields, it is essential to consider what settings might best serve the largest group of people.

2.1.1 The Medical Field

In the medical field, the integrity of data, particularly MRI information, is of the utmost importance. MRI data is sensitive and a foundation for critical diagnoses and treatment plans. A signed plugin is essential to ensure the filter for such data remains unaltered and reliable. This signature is a safeguard, providing the results are trustworthy and free from malicious or accidental tampering. The need for accuracy in analysis concerns not only data but also patient safety.

Furthermore, many healthcare institutions develop proprietary filters specifically for processing MRI scans. By signing these filters, these institutions can protect their intellectual property, preventing unauthorized copying or modification, thus preserving their competitive edge. Additionally, signed filters create an audit trail, allowing for verification in case questions arise regarding the analysis results. This traceability is crucial for regulatory compliance, such as adhering to HIPAA regulations and maintaining legal defensibility. For instance, if a hospital utilizes a proprietary filter to eliminate noise from MRI scans, signing that filter ensures that only validated versions are in use by radiologists. This step is vital in preventing the potential pitfalls of altered images that could lead to misdiagnosis.

2.1.2 The Financial Field

The need for data protection takes on a different hue in the financial sector but is equally critical. Financial institutions often rely on complex algorithms and filters for trading, risk assessment, and fraud detection—highly confidential elements representing a significant competitive advantage. Here, signed plugins become instrumental in safeguarding these proprietary filters from unauthorized access and potential data leaks. The integrity of financial models hinges on the authenticity of the data and the filters applied. If a filter is tampered with, it could lead to erroneous risk assessments and flawed trading decisions, resulting in substantial financial losses. Signed plugins guarantee that the filters used are genuine and uncompromised.

Additionally, these institutions must navigate stringent regulations concerning data security and financial reporting accuracy. By providing clear evidence of trustworthy filters, signed plugins assist in demonstrating compliance with these regulations. For example, a hedge fund might develop a proprietary filter for analyzing market trends. By signing this plugin, the fund ensures that only authorized personnel can access it, thus protecting their valuable analysis methods from competitors.

2.1.3 Government Security

Data security and integrity are paramount in government and high-performance computing (HPC) environments, such as those encountered by agencies like NASA. These agencies handle highly

sensitive information, and the implications of unauthorized modifications could compromise research results or national security. In HPC contexts, where complex simulations and data processing tasks are commonplace, signed plugins are crucial in ensuring that the filters employed are genuine and untampered with. This commitment to integrity guarantees the validity of computational results. Signed plugins offer a verifiable record of the filters used, contributing to that chain of custody. As an illustration, NASA might utilize HPC to simulate climate models; here, signed plugins ensure that the filters applied to climate data remain unaltered, thereby securing the accuracy of the model predictions.

2.1.4 Cloud Storage

The balance between scalability, cost-effectiveness, and security in cloud environments poses unique challenges. The significance of signed plugins is heightened in the cloud, where data and processing resources are frequently shared among various users, increasing vulnerability to a wider array of security threats. The Hierarchical Data Service (HSDS), often used for managing large scientific datasets within cloud infrastructure, can greatly benefit from signed plugins. These plugins ensure that only verified filters are applied to the data stored within the HSDS framework.

Additionally, signed plugins can enhance access control mechanisms by linking filters to specific user identities and permissions, allowing administrators to manage which users can utilize which filters effectively. Maintaining data provenance is also vital in cloud environments; signed plugins contribute valuable records of filter applications to data, enabling researchers to trace how the data has been transformed. For example, in a research institution storing genomic data within HSDS, signed plugins would guarantee that only approved filters are employed, preserving the integrity and confidentiality of the research process.

2.2 Options for General Levels of Authentication

Most users will likely want to use it; however, if they wish to use both signed and unsigned plugins, we may need to provide various access levels—high, medium, low, and off—to give them greater control.

- **Off** – When this option is selected, the feature becomes inactive, allowing users to utilize any plugin as they did before without restrictions. This is especially relevant for users who are less concerned about security and prefer to continue using HDF5 as they currently do. Furthermore, it may attract users who do not want to install GPG-related packages due to a lack of knowledge about library installations.
- **Low** – Selecting this option enables the use of both signed (though not verified) and unsigned plugins. This targets users that are interested in the new feature but who are not opposed to using unsigned plugins. It facilitates an easier transition for these users since they won't be restricted to using only signed plugins.
- **Medium** – This setting allows only signed (but not verified) plugins. It is designed for users who have a reasonable level of trust in the sources of these plugins. This option is useful for

those who might have outdated versions of plugins that, although not verifiable, are still signed and can be safely used.
- **High** – This option allows only signed and verified plugins. It targets users who prioritize security and wish to reduce the risk of third-party data manipulation or theft. This setting may be particularly advantageous for institutions, as mentioned earlier, that require highly secure and private data storage.

3 Implementation and Technical Details

The primary focus will be on utilizing GPG-related tools as a proof of concept for implementation to assess the feasibility and user experience of the proposed approach. This approach does not exclude exploring alternative signing mechanisms and key management strategies to discover potential solutions.

To use GPG, this guide utilizes "*GnuPG Made Easy*" **(GPGME)**, which is available on all leading Linux distributions and simplifies access to GnuPG for applications. GPGME is a library that allows GPG to be used programmatically without the need for the command line. **Gpg4win** is a Windows-specific package that features a GUI. Other GUI-based alternatives include **Kleopatra** for Windows and **GPGTools** for macOS.

Since the aim of this effort is to increase security and trust in HDF5 plugins ultimately, it requires a few amendments to how users might typically use HDF5:

- Users must first determine if signed plugins are necessary. If not, no changes are required for the workflow, and users can continue to use HDF5 as they previously did.
- If signed plugins are in HDF5, the CMake option HDF5_REQUIRE_SIGNED_PLUGINS should be enabled to require digitally signed plugins when building the HDF5 library.
- With GPGME installed system-wide, CMake's "FindGpgme" should be able to locate the software. Otherwise, the paths to the software's installation location can be passed to CMake.
- Build HDF5 as needed with the necessary options.
- Install the plugin of interest.
    - Download the signature and public key and find the ID number for the plugin.
- The plugin path must be set up in HDF5 along with the public key, signature, and plugin ID.
- The public key needs to be verified and signed to be used to verify the signature and plugin.
- HDF5 must verify that it is a properly signed plugin and can be used.
- The user may proceed as usual.

3.1 HDF5 Build System Changes

To implement this feature, a new build option will be added to enable or disable it. Additionally, to implement the differing levels of privacy for this option, an environment variable will be added, HDF5_PLUGIN_PRIVACY_LEVEL, that takes "high", "medium", "low", and "off" as options. This will allow users to adjust the privacy level as necessary, even if they didn't build directly from source code.

A new build option, HDF5_REQUIRE_SIGNED_PLUGINS, will be created to toggle the option to require digitally signed plugins for CMake. This will use the CMake to link to the gpgme and gpg-error libraries to find the required header file locations. Ifdefs will be added around any new code that will enable and disable the new feature using the new build option to control a global variable.

To begin, adjustments need to be made to HDF5's *CMakePlugins.cmake* file to introduce the new configuration option. This option should remain set to OFF by default.

```
#-----------------------------------------------------------------------------
# Option to Require Digitally Signed plugins
#-----------------------------------------------------------------------------
option (HDF5_REQUIRE_SIGNED_PLUGINS "Require digitally signed plugins" ON)
if (HDF5_REQUIRE_SIGNED_PLUGINS)
  if (NOT WIN32)
    add_compile_definitions(H5_REQUIRE_DIGITAL_SIGNATURE)
    link_libraries(gpgme)
    link_libraries(gpg-error)
  endif ()
endif ()
message(VERBOSE "Digital signatures required for all plugins")

```

3.2 HDF5 Library Changes

The current implementation method requires changes to the plugin code, which is primarily found in H5PL. The current approach is to add the plugin signature verification code to H5PL__open in H5PLint.c.

An approach that utilizes environment variables could also be considered. In this case, setting adding an environment variable PL_REQUIRE_DIGITAL_SIGNATURE in H5PL__init_package would toggle the above-mentioned global variable HDF5_REQUIRE_SIGNED_PLUGINS. This would occur after checking whether to load plugins using the same function. If the environment variable is set, we would enable the global variable and proceed as usual.

Alternatively, the verification code could be included in the HDF5 function H5PL_load, which checks whether plugins can be loaded for the current plugin type. This would add another boolean to the check, H5PL_require_digital_signature_g. Since H5PL_load calls H5PL__open, both methods should work.

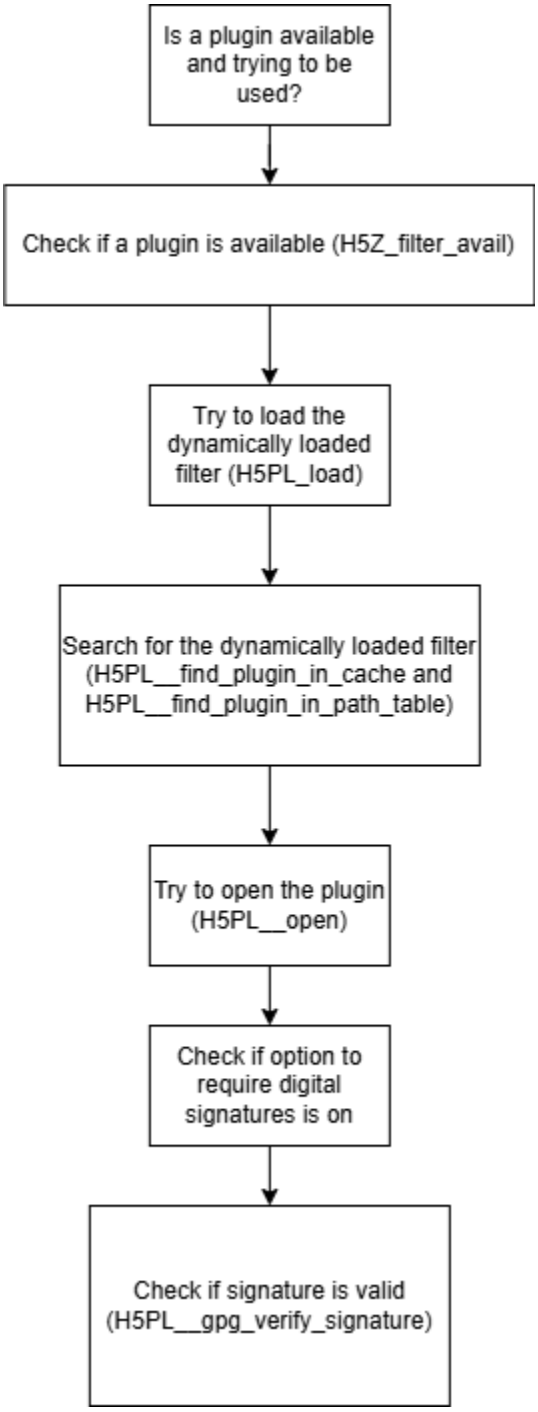

Figure 1 Process for loading the signed filtered plugin.

Figure 1 depicts the plugin loading process. When a file is read and plugins are enabled, the library invokes H5PL_load, which first checks the cache for the filter plugin. If it is located, it should have already loaded and verified the plugin earlier, thus allowing its usage.

If the plugin isn't found, it invokes `H5PL__find_plugin_in_path_table`, which checks the plugin against each path in the table and then calls `H5PL__find_plugin_in_path` on it.

H5PL__find_plugin_in_path opens the directory and iterates through it, searching for the dynamic library. Once located, it calls H5PL__open to try to open the plugin.

After obtaining the plugin type and information, the new code for checking if a valid digital signature exists should follow. This code will be enabled or disabled based on a global variable controlled by the build option HDF5_REQUIRE_SIGNED_PLUGINS. Upon verification, if the signature is valid, this code should resume previous behavior and continue attempting to load the plugin. If the signature is invalid, it should throw an error.

3.3 Factors for VFDs and VOLs

For VFD plugins, VFD creators currently need to link to a relevant version of the library, implement two functions, H5PLget_plugin_type and H5PLget_plugin_info, and provide a VFL-compliant "class" structure for the VFD. If digital signatures are required, when the VFD calls H5FD_register_driver_by_name, which registers a new VFD in the library using its name, it calls H5PL_load and specifies that the plugin type is H5PL_TYPE_VFD. When this code is executed, run the new code first, and then proceed to the switch statement to open a plugin of type VFD.

For VOL connector plugins, if digital signatures were required, when the VOL connector calls H5VL__register_connector_by_name, which registers a new VOL connector using its name, it calls H5PL_load and specifies that the plugin type is H5PL_TYPE_VOL, and calls the same code as mentioned above. The current implementation should work for VOL connectors as well.

3.4 Other Factors

Another consideration for this approach is that GPGME can be used in a multi-threaded environment; however, it is not entirely thread-safe and requires special support. For this use case, the safest option would be to use GPGME on a single thread and transmit the result to other threads.

Additionally, there may be a need to support files over 2 gigabytes, and further provisions will be necessary to accommodate larger file sizes. Although GPGME has large file support by default, as long as the system it runs on can support it, certain systems can only support up to two gigabytes. In those cases, largefile support will need to be enabled in GPGME so that it will work as expected. In this case, depending on the system, we may have to add H5PL__check_supports_largefile to check and H5PL__handle_largefile to add explicit support.

While the initial implementation targets Linux, additional modifications are necessary for compatibility with Windows and MacOS. As it stands, attempting to use the feature on Windows or MacOS will lead to errors.

4 Further Considerations

Enhancing security practices by regularly assessing the centralized database and resources for plugin maintainers is crucial for further considerations in plugin management. Establishing user feedback mechanisms and conducting code reviews will strengthen defenses against malicious plugins. Additionally, fostering community engagement and providing support during the transition will help ensure all stakeholders can adapt effectively to these vital security measures.

In the case of a closed system, like a classified network with no internet connection, preliminary findings indicate that GPG will not be able to function. GPG uses the internet to access public key servers to look up and validate keys, managing keys, or getting key data. More research will be required to

4.1 Storage and Management of Public Keys, Signatures, and IDs

When considering the storage and management of public keys, signatures, and IDs associated with each plugin, a centralized database managed by a trusted authority, such as the HDF Group or another designated body, emerges as the most robust long-term solution. This approach allows for automated updates, version control, and more straightforward revocation compared to traditional methods.

Although the initial setup may be more intensive, this system scales better than relying on manual updates or user-uploadable databases, which could introduce significant security risks and management overhead. Ideally, this centralized database should link each plugin ID to its corresponding public key and signature information.

4.2 Revocation and Updates

Revoking compromised plugins and distributing secure versions requires a clear strategy. Revocation should involve updating the central database to mark any compromised plugins as invalid, coupled with announcements and pointers to new, secure versions. When updating plugins that necessitate re-signing, updating the database with the latest signature and, if applicable, a new public key is crucial. Version control within the database is essential to prevent conflicts, allowing clients to verify the latest valid signature and public key before executing a plugin. Utilizing timestamps or version numbers associated with signatures can help manage the validity across different versions.

4.3 Responsibility for Updates

The plugin maintainers are primarily responsible for updating plugin information, especially for older plugins. They should take charge of updating their respective plugins' information, including re-signing after any updates. While a central authority can provide necessary tools and documentation to aid this process, outdated or unmaintained plugins may require intervention from the HDF Group or designated community members to ensure timely updates or possibly consider their deprecation.

4.4 Handling Malicious Plugins with Valid Signatures

Addressing the risk posed by malicious plugins with valid signatures necessitates a multi-layered approach since digital signatures confirm neither behavior nor intent. Measures such as code reviews or analyses should be encouraged, particularly for plugins sourced from untrusted origins. Running

plugins in a sandboxed environment can further limit their access to system resources, mitigating potential damage. Additionally, facilitating a user reporting mechanism for suspicious plugin behavior, alongside developing a reputation system based on user feedback and code analyses, can enhance overall security.

4.5 User Interaction with Keys and Signatures

Automation can streamline the user experience regarding the public key and signature file specification. The system should automatically discover these necessary files based on conventions such as naming conventions like *plugin_name.signature* and *plugin_name.pub* in respective directories. Should automatic discovery fail, users should have the option to specify file locations manually. However, the primary method should prioritize the database lookup for efficiency and effectiveness.

4.6 Transition Period

Implementing digital signatures for plugins will require a well-thought-out transition process. A phased approach is recommended, starting with a clear announcement of the introduction of digital signatures alongside comprehensive documentation and tutorials for plugin authors. Allowing a grace period gives authors time to sign their plugins and submit the necessary information. Highlighting signed plugins in documentation or on the website can incentivize early adoption. Gradual enforcement will ensure that digital signatures become mandatory for all new plugins and, eventually, for updates to existing ones. Lastly, providing support channels for plugin authors encountering difficulties during the transition will facilitate smoother adoption.

4.7 Trusted Sources

Another aspect to consider is a system for plugin distributors similar to Microsoft's Trusted Developer Program. In this system, developers must register with Microsoft and attain trusted status. This process includes account verification, possible background checks, and reviews of their code or apps. Once this status is obtained, users can distribute their software through the Microsoft Store. The Apple Developer Program operates similarly. Additionally, Apple has Gatekeeper, which is a version of the digitally signed plugin feature we aim to implement here. Gatekeeper safeguards against malicious software by ensuring that apps originate from the Apple Store or are signed with a valid developer certificate from Apple.

5 Testing

Proper testing is essential to ensure this feature works correctly. For this purpose, a test plugin, VFD, and VOL connector will need to be selected or created. Candidates include the NullVFD and NullConnector. Furthermore, we have several test plugins already available. Once a test plugin is identified, a real key should be generated. This key will be used to sign the test plugin, creating a legitimate signature file. Additionally, we must create an improper signature file. Both signature files should be tested.

Creating continuous integration workflows for ongoing testing of this feature in GitHub would also be beneficial. Numerous permutations must be tested thoroughly, requiring a nightly workflow rather than the ones that trigger automatically upon code pushes. For instance, testing scenarios where a plugin is unsigned, a plugin is signed, but the signature is invalid, and the situation where a plugin is signed with a valid signature must all be evaluated against the various security levels discussed earlier in the RFC. This likely necessitates several rounds of testing, which may be time-consuming.

6 Acknowledgments

This work was supported by the National Science Foundation under Grant No. GR136407 to The Ohio State University, with a subaward to The HDF Group under Subaward No. 2419722.

7 Revision History

January 14, 2025: Version 1 circulated for comment within The HDF Group.

8 Appendix: Examples

Examples for how this would work.