Software Identification Ecosystem Option Analysis

October 2023
Cybersecurity and Infrastructure Security Agency

Executive Summary

Organizations of all sizes must track what software they own and operate to perform user support, inventory administration, and vulnerability management.

Effective vulnerability management requires software to be trackable in a way that allows correlation with other information such as known vulnerabilities, mitigations for those vulnerabilities (e.g., available patches), lists of approved or disallowed software, and adversary activities. This correlation is only possible when different cybersecurity professionals know they are talking about the same software.

Software identifiers are labels for specific versions of software that conform to a defined format. Several unharmonized identification solutions exist, resulting in the same software products and their versions having inconsistent identifiers across the ecosystem. An effective, harmonized software identification ecosystem will facilitate greater automation, inventory visibility, and broader, more effective use of software bills of materials (SBOMs).

The two key requirements for an effective software identification ecosystem are:

1. Timely availability of software identifiers across all software items
2. Software identifiers that support both precision and grouping

Some existing software identifiers do a good job of addressing one of these two requirements. However, no existing software identifier adequately meets both requirements. CISA and HSSEDI (Homeland Security Systems Engineering and Development Institute)[1] experts surfaced the following features that help a software identifier ecosystem meet these requirements:

- Any identifier scheme should include properties common to the SBOM and vulnerability management use cases, such as software name and version.
- An authority that establishes common rules, assigns responsibilities, and identifies and addresses issues for the identifier generators would likely improve the overall accuracy and robustness of the software identifier ecosystem. Such an authority would need to be securely funded. Without a global authority, the operational effectiveness of the ecosystem will depend on the individual efforts of the identifier generator community.
- While inherent identifiers provide optimal availability, inherent identifiers that support grouping remain an unsolved engineering challenge.

An adequate software identifier ecosystem consists of identifier formats in concert with operational entities—such as a central authority and identifier generators. Three software identifier formats are the most promising as starting points:

- **Common Platform Enumeration** (CPE) would provide a basis to pursue **Defined Identifiers with Grouping Expressions**. CPE's challenge will be evolving it to a more distributed production model to address scalability and sustainability challenges.
- **OmniBOR** provides a strong starting point to pursue **Inherent Identifiers**. The biggest challenge will be fitting OmniBOR identifiers to the varying granularities of software data artifacts that need to be annotated.

[1] For more information on HSSEDI, see https://www.mitre.org/our-impact/rd-centers/homeland-security-systems-engineering-and-development-institute

- **Package URLs** (purl) is a promising **Defined Identifier** that could support either **More Comprehensive Enumeration Support** or **Grouping Expressions**. The challenge the purl specification faces is adoption in software development communities besides package managers.

An adequate, relatively quick solution to a comprehensive software identification ecosystem may be to use each identifier format where it is most effective. This approach would require establishing a multi-identifier ecosystem with clear rules defining which identifier is used under which conditions. Making such clear rules and establishing strong norms to follow them satisfies the core challenge to reach this solution. Each software item would have exactly one identifier, but the format of that identifier may be different to best suit the needs of the relevant stakeholder communities.

Table of Contents

1 THE PROBLEM

Organizations of all sizes need to track what software they own and operate to perform user support, inventory administration, and vulnerability management.

Effective vulnerability management requires software to be trackable in a way that allows correlation with other information such as known vulnerabilities, mitigations for those vulnerabilities (e.g., available patches), lists of approved or disallowed software, and adversary activities. This correlation is only possible when different cybersecurity professionals know they are talking about the same software. Several unharmonized identification solutions exist, resulting in the same software products and their versions having different identification schemes across the ecosystem.

The vulnerability management landscape cannot transform through greater automation, inventory visibility, and the multifaceted value proposition of software bills of materials (SBOM's) broad adoption without a harmonized software identification scheme easy to adopt and scale. This paper characterizes the requirements and activities necessary to support that goal.

1.1 Goal and Audience

A more harmonized software identifier ecosystem is necessary to support data correlation activities for critical use cases. This paper uses the term "software identifier ecosystem" in recognition of the fact that the success of any software identification scheme requires more than just the definition of an identifier format, but also requires broad production and adoption of identifiers, clear processes, and (in some cases) robust supporting infrastructure. Successful realization of such an ecosystem would enable more efficient and accurate software management for security, policy, and supply chain management.

Our goal is to catalyze community discussion and action. We do not dictate a solution, but rather presents a set of possible paths forward to address the key challenges standing in the way of a more robust software identifier ecosystem. Each path has associated costs and tradeoffs.

This paper focuses on an ambitious goal: a software identification ecosystem that can be used across the complete, global software space for all key cybersecurity use cases. While no software identification scheme today fully addresses this objective, many schemes do address subsets of this overall problem space. For example, package managers and related identification schemes such as Package URLs (purls) have been used within package managers. While there are challenges in extending these schemes to support different parts of the global software space—namely, software that is not distributed via managed packages—and use cases, such as mapping to vulnerability reports, this prior work creates a valuable foundation to build upon.

1.2 Related Work and Methodology

For a description of existing work related to this paper, see 0.

1.3 Key Requirements

A software identifier ecosystem must enable correlation across datasets, such as an organization's software inventory and a database of known vulnerabilities.

There are two key requirements for a software identifier ecosystem to enable correlation:

1) **Make identifiers available when and where needed**—Correlating data sets using software identifiers requires that both data sets annotate their data artifacts with the appropriate identifiers. This means that the necessary identifiers need to exist when data artifacts are created and that data artifact creators need to know what those identifiers are. For example, when an inventory tool discovers a particular application on an endpoint, it must be able to discover that application's software identifier to annotate the associated finding in its inventory. Likewise, when a vulnerability researcher creates a

record describing a vulnerability for a specific piece of software, they will need that software's identifier to tag this record. Only records annotated with software identifiers can be correlated via automation. Therefore, the first requirement is that identifiers are available when and where they are needed.

2) **Support granularity of data artifacts**—Different artifacts deal with software at different levels of granularity. For example, a finding from an inventory scanner will (ideally) identify the specific variant of a discovered piece of software. By contrast, a vulnerability report such as a CVE Record might apply to a range of versions and variants of software. To ensure the value of correlations, the identifier(s) used to annotate a data artifact must exactly match the scope of the software the artifact describes. This requires that identifier formats support both precise (i.e., a single software variant) and broad (i.e., a large range of software variants) software identification expressions. To overcome the challenges in doing this effectively, the software identifier ecosystem must provide support.

Each key requirement is treated separately and includes several potential paths; different mechanisms serve different requirements, and there are a variety of ways to mix and match the different requirement paths. The paths presented for each key requirement focus on ways to enable a robust correlation capability using software identifiers. Paths serving a given requirement might not be mutually exclusive but are intended to represent alternative solutions to the same problems. Paths supporting different requirements may align with or complicate each other; these situations are noted in the path descriptions.

2 KEY REQUIREMENT: MAKE IDENTIFIERS AVAILABLE WHEN AND WHERE NEEDED

Every data set (for example, those created by an enterprise inventory tool or by a vulnerability researcher) needs to have immediate access to the correct identifier for the piece(s) of software in question to tag artifacts appropriately. This has been a challenge for some historic software identification standards. For example, common platform enumeration (CPE) identifiers are often not created until after vulnerabilities are discovered in a piece of software. As a result, the initial vulnerability report is generally unable to include a CPE identifier and cannot support correlation based on CPE identifiers.

Even when identifiers exist before they are needed, would-be users require access to those identifiers. While some identifier schemas have a built-in mechanism by which identifiers are made available with the associated software (for example, Universal Bill of Receipts (OmniBOR) and software identification (SWID) tags), others lack a simple process by which to learn the identifier for a piece of software. When determining a piece of software's identifier is difficult, time-consuming, and/or error-prone, it reduces the chance that data artifacts will be correctly tagged with their associated identifiers.

The following paths describe ways to develop a process to ensure identifiers are available to tag data artifacts with software identifiers. The distinguishing feature between the paths lies in when and by whom the identifiers are generated. In one path, called "Inherent identifiers," identifiers can be generated by any party at any time. This is possible because these identifiers are mechanically generated based on the inherent properties of a piece of software available to anyone possessing that piece of software. In the second path, called "Defined identifiers", identifiers are created only by certain parties and only at one point in time. These parties assert an association between an identifier and some piece of software, and then must publish this association so that others can use it. Examples exist of both types of identifiers in use today. Both types of identifiers have advantages. However, both types of identifier schemes also face challenges not fully addressed by the identifier ecosystems currently in existence.

2.1 Path 1: Inherent Identifiers

An inherent identifier is a software identifier that any party can deterministically derive from an instance of a piece of software. For example, OmniBOR (formerly GitBOM) creates identifiers from the SHA1 or SHA256 hash

of file information. The key criterion for inherent identifiers is that there is a mechanical process that any party can perform. An identifier scheme is not "inherent" if knowledge is dependent upon certain parties or if different parties have different understandings of the input.

While simple in principle, inherent identifiers face two main challenges, discussed in the following paragraphs

First, it isn't always obvious which file(s) should be the input to the identifier generation function for large, multi-file applications. Large applications often have a "launcher" executable file that is the front-end to multiple other executables and libraries. However, when an application gets a new revision, the launcher does not always change, so if the application launcher is the only input to the identifier generation function, multiple versions of the same piece of software might get the same identifier. This can be addressed by using multiple files as the input to the identifier generation function, but defining which files should serve as inputs in a way that all parties will understand can be difficult.

Large applications can involve dozens, or even hundreds, of files; some of which will be specific to each install (and thus should be excluded if the goal is to create a common identifier across installations), and some of which might represent libraries shared by multiple applications. To ensure all parties create the same software identifier for the same piece of software, there must be a reliable method of ensuring that all parties always use the same set of files as inputs, while ensuring that any files that may differ between installations of the same variant of the software are excluded. For its part, OmniBOR focuses on providing an identifier for individual files, but uses a separate manifest to capture collections of related files, and the standard does not attempt to normalize manifest inputs. As such, while OmniBOR provides a universal and globally unique identifier for individual files, it does not necessarily provide a universal and globally unique identifier for software applications.

The second challenge is that most of the properties that describe and organize software are not "inherent." People discuss software using its name, vendor, version, and a few other properties; but these are labels that some party applies to the software, rather than inherent properties of software files. The result is that inherent identifiers are often not easily understood or used by human readers. More significantly, many types of data artifacts against which correlation activities would be desirable refer to groups of software that are delimited by these non-inherent properties. For example, many software vulnerability reports apply to a range of versions for a particular piece of software. Because today's inherent identifiers don't capture the properties that define common software groups, it can be challenging to use them to label data artifacts that apply to such groups. Section 3.1 discusses the need to match identifiers to the granularity of artifacts in more detail, including noting the necessary identifier properties and ecosystem infrastructure needed to do this. For the current discussion, it is sufficient to note that it can be difficult for inherent identifiers to address this requirement.

2.1.1 Actions to Pursue this Path

Address unsolved challenges for inherent identifiers: The authors are not aware of any existing inherent identifier scheme that adequately addresses either of the challenges. As such, a format would need to be developed (or adapted from an existing format) that addresses these challenges. Developing such an identifier scheme is arguably a more challenging task than developing defined identifiers (described in the rest of this Section). The latter only needs to support identifier generation for a small set of parties who may have specific skills and knowledge, while the process to generate an inherent identifier scheme must be usable by any party in a wide range of circumstances. As such, this represents an open research question.

2.1.2 Expected Path Outcomes

In many ways, it would be an ideal solution for someone to devise an inherent identifier scheme that addresses the key challenges noted above. An effective inherent identifier scheme means that an identifier would always be available to any party with access to the software in question, which covers most use cases. No party bears the resource cost of creating identifiers for the entire ecosystem, because everyone should be able to easily

create the identifiers they need. Identifier discovery is streamlined because personnel can create identifiers as needed, rather than looking them up from other sources. Likewise, there would be no issues around the sufficiency of identifier coverage, as there would be no software for which identifiers personnel could not create identifiers. Even legacy or custom, in-house applications would have a unique identifier that users could use in any context. Defined identifiers (Sections 2.2, 2.3, and 2.4) struggle with adequate production rate, identifier discovery, and coverage. Inherent identifiers provide a way to overcome these issues, which makes them an attractive option. However, inherent identifiers struggle to meet the second key requirement of supporting flexible identifier granularity. Figure 1 shows the cycle of identifier creation and use for inherent identifiers. Notably, this entire cycle happens with the party using the identifier (i.e., the party seeking to learn the identifier for a piece of software in order to label some software data artifact).

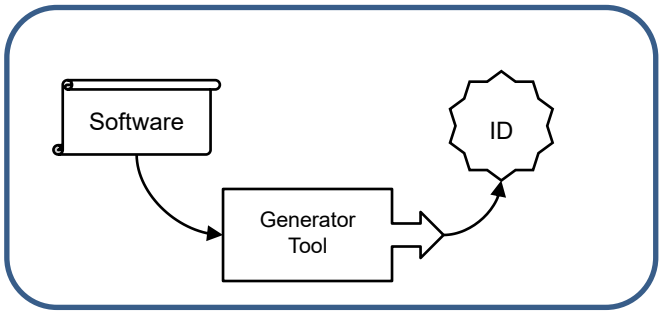


Figure 1: Inherent Identifier Creation and Use

Background on Cluster of Paths Around Defined identifiers

A defined identifier is an identifier that some party declares is associated with a particular piece of software. A designated party asserts the binding between identifier and software. Other parties wishing to use an identifier for a given piece of software need to be informed of the associated identifier. Defined identifiers are the alternative to inherent identifiers.

There are many existing defined identifier formats, including CPE, purls , and SWID tags. As defined identifiers are associated with a piece of software by fiat, they take on a wide variety of formats: CPEs consist of several fields reflecting properties of the software in question, SWID tag identifiers that are Globally Unique Identifiers (GUIDs), and purls that are uniform resource identifiers (URIs) whose structures vary depending on the software's package management system.

Current defined identifier formats face two significant challenges in creating a broad and robust software identifier ecosystem:

1. Some party or parties need to take on the role of asserting the bindings between identifiers and software. This paper refers to such parties as "identifier generators." Because a piece of software does not have any software identifier until an authority asserts one for it, identifier generators need to assert these bindings at a rate that, ideally, keeps up with the rate at which software variants (versions, editions, etc.) are produced, or at least the rate at which they are referenced in data artifacts. Within certain environments, this has been achieved. For example, package management systems automatically provide a unique identifier for every package within their purviews. However, software identification schemes that cover all global software continue to struggle with providing sufficient, timely coverage.
2. The parties that wish to tag artifacts with software identifiers need a way to learn the identifier associated with a given piece of software. There are different ways that this can be accomplished:

SWID tag identifiers travel with the software itself, while CPEs are stored in a central, searchable database. Regardless of the method, there must be some low-effort mechanism that allows one to reliably learn the identifier for a given piece of software.

Because of these challenges, the success of a defined identifier scheme hinges less on the structure of the identifier than on the overall process for creation and propagation of the identifier. As such, the paths outlined below focus on identifier production and propagation rather than format details. The central difference between the first two sub-paths presented below concerns whether the model avoids active management of the ecosystem and allows identifier production without oversight (Section 2.2) or if the model employs some authority to manage the identifier space (Section 2.3). There are community concerns about the burdens a central authority could create for identifier generators, as well as concerns about the potential of central authorities to become bottlenecks within the software identifier ecosystem. In both cases, it is expected that there is a software identifier format owner who publishes and revises the specification defining the identifier format and processes. However, this is different from a party that is part of the day-to-day operation of the ecosystem, which would be the role of an authority in the model. It is the need for an active authority in the ecosystem that differentiates the two models presented in Sections 2.2 and 2.3.

The remaining sections discuss nuances in selecting or compromising between the options laid out in Sections 2.2 and 2.3. Section 2.4 highlights that the centralization-decentralization of the authority enforcing defined identifiers is not all or nothing. Section 2.5 discusses how to create a backstop for the inevitable situation in which a piece of software is (erroneously or due to immature processes) not provided with an identifier. Section 2.6 clarifies that multiple identifier formats is a distinct and separate issue from the topic of identifier providers and identifier authorities.

2.2 2.32.42.52.6Path 2: Unmanaged, Distributed Model

In the model this path describes, many parties generate identifiers without oversight or coordination. The idea of using many parties to create identifiers is intended to maximize software identifier ecosystem coverage while spreading out the work so that no one party bears a disproportionate burden for identifier generation. Requirements for oversight and coordination can create significant overhead for identifier generators. As such, avoiding such requirements should avoid the associated burdens, hopefully leading to greater identifier generator participation.

2.2.1 Actions to Pursue this Path

Establishing an ecosystem where this model is possible creates several requirements on identifier formats, creators, and users:

Generator-specific markings in identifiers: The identifier format must define a way for all identifier creators to specifically denote the identifiers they create. The denotations must be reconciled to ensure that two different identifier creators will never accidentally assign the same identifier to different pieces of software. Such an occurrence (called “identifier reuse”) could lead to false-positive correlations and must be avoided. Ensuring identifier formats have this ability would be the job of the software identifier format owner.

Clear division of the software space among generators: The converse of identifier reuse is called “overidentification” and occurs when multiple software identifiers are created for a single piece of software. Overidentification is problematic because it leads to false-negative correlations if two data artifacts for the same piece of software are tagged with different identifiers. To minimize this occurrence under this model, the global software space would need to be divided among identifier generators, so no two generators create identifiers for the same piece of software. This would need to be done in a way that was understood by all generators without the need for some central authority to explicitly define each generator’s coverage. One example of such a division would be if a software creator or publisher was expected to create identifiers only

for the software they create/publish. The identifier format owner would likely define how the software space was divided among identifier generators.

Push identifiers with software: After identifiers are created, there must be a way for other parties to learn them so that data artifacts can be correctly tagged. In a highly distributed identifier generation environment, having each identifier generator host a database with which personnel can look up identifiers is not practical—hosting such a service reliably may be beyond the means of some identifier generators, and it creates challenges for users who would need to figure out which database to use to find a given identifier. A central database of identifiers is only practical if all identifier generators are pushing their identifiers to this database, but this means there must be coordination with a central authority, the avoidance of which is the primary goal of this model. Instead, for users to obtain identifiers would likely be to push them out with the software with which they are associated. This could be done by embedding identifiers in the software files themselves (as is done by OmniBOR) or including the identifiers in ancillary files that are distributed with the software (as is done in SWID tags). Such a model would require the identifiers to exist at the point of software distribution, which would be another reason for software creators and publishers to also be the software identifier source. Inclusion of software identifiers with software would be the job of software publishers and distributors.

Minimize required information in identifiers: The most critical factor for this model will be getting enough parties to generate accurate identifiers to cover a critical mass of the global software space. Every piece of information the identifier format requires potentially increases the workload for identifier generators and creates opportunities for errors. For this reason, entities establishing identifier formats should minimize the amount of information generators require to collect and embed in identifiers.

Incentivize identifier creation: Another way to encourage identifier generation in the hope of achieving a critical mass is to offer incentives for identifier generation and penalties for parties that distribute software without identifiers. This will likely involve requiring the presence of identifiers as a prerequisite for entry into certain markets. For example, app store owners may require all apps they publish to come with identifiers, or certain large-scale buyers might require identifiers for all their purchases. Creating such expectations would further encourage parties to support identifier generation. Parties that serve as gatekeepers to markets (app store owners, parties defining purchasing requirements for large organizations, etc.) would need to be the ones to enact this requirement.

2.2.2 Expected Path Outcomes

This model aims to maximize coverage of the global software space by leveraging as large a cadre of parties as possible creating software identifiers. This is facilitated by minimizing the burdens associated with software identifier generation (removal of the burdens of external coordination and simplification of identifier generation) while creating incentives for cooperation. The degree to which the software space is covered will be the biggest challenge for this model and its primary measure of success.

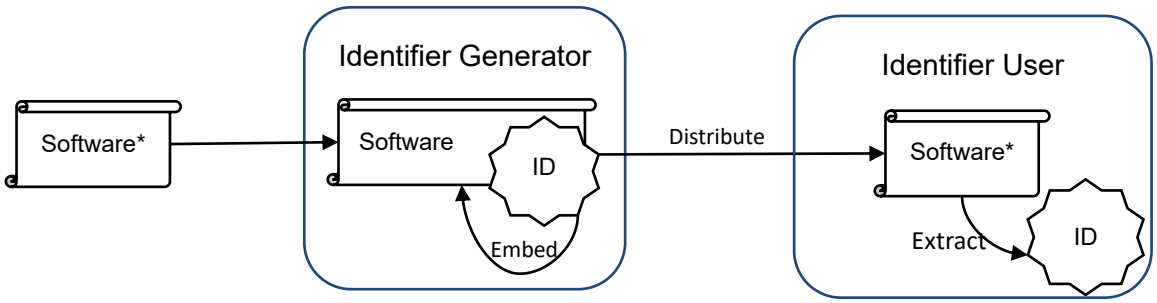
Assuming the coverage of the global software space is adequate, creating and pushing identifiers with software items would ensure identifier availability timeliness. As noted earlier, a common complaint about the CPE identifier scheme is that identifiers are generally created only after there is a demonstrated need, which is too late for their inclusion in initial data artifacts. By contrast, because identifiers are created at the software source in this model, they would be co-temporal with their associated software, thus eliminating this time gap between need and creation.

The quality of the software identifiers is one potential challenge in this model. For software identifiers, quality issues might manifest as inconsistency in how certain information is captured (e.g., the same identifier generator using different names for the same company, which has been seen in some CPE identifiers) or as failure to follow granularity requirements of the standard by lumping multiple variants of software under a single identifier. Clear guidance to identifier generators will be important to reduce quality issues, as will the

presence of standardized tools for automated identifier generation. This will, however, require up-front investment in these supporting mechanisms.

Unfortunately, complete coverage of the global software space with defined identifiers is not plausible, as there will always be software whose creators do not participate in identifier creation. Likewise, legacy software (i.e., software that existed before a chosen identifier scheme was deployed) might never receive an identifier if its creator does not take the effort to create identifiers for all its old software. As such, it is virtually inevitable that this model will lead to incomplete coverage of a typical enterprise's software. Having parties create ad hoc identifiers for unidentified software is not a viable strategy because ad hoc identifiers are of limited value for correlation and would likely lead to overidentification of software. Section 2.4 describes a supplemental mechanism to characterize (rather than create an identifier for) such software, which would somewhat mitigate the issue of unidentified software.

Figure 2 shows the process for identifier creation and use for unmanaged, distributed identifiers. In this model, the identifier generator is the source of both the software and its identifier. (There could be intermediate parties involved, but these are not shown in the figure.) "Embedding" the identifier with the software could take the form of embedding the identifier within an executable (such as OmniBOR) or including it in a separate file that is distributed with the software (such as SWID tags). The result is that identifier user possession of the software directly leads to the user being able to learn that software's identifier.



*: Software might be the Identifier Generator's own or from a third party, whose software the Generator distributes.

Figure 2: Unmanaged Defined Identifier Creation and Use

2.3 Path 3: Managed, Distributed Model

In the model this path describes, a central authority supports and coordinates the activities of multiple software identifier creators. The duties of the authority could include some combination of the following:

- Assigning responsibility to create identifiers for certain sets of software.
- Providing a searchable central repository of identifiers.
- Identifying issues with the identifier space (e.g., identifier reuse, overidentification, critical gaps in coverage) and reconciling them.

Like the model in section 2.2, this model relies on a distributed community of identifier generators. However, in this model, there is a central authority that oversees those activities. The authority supports the community by helping to ensure quality, availability, and sufficient coverage of identifiers. This, however, requires identifier producers to coordinate with the authority, which increases the burden on identifier generators. Automation could help reduce this burden on generators to nearly nothing, but it may still deter some parties from

participating as identifier generators. Likewise, the authority itself will need to be provided with sufficient resources to avoid becoming a bottleneck for the community.

2.3.1 Actions to Pursue this Path

This model shares some requirements with the unmanaged model in section 2.2, but also has some key differences:

Generator-specific markings in identifiers: While this model theoretically makes it possible to manage the identifier space without generator-specific markings in identifiers, managing an identifier space without such markings requires additional coordination between the central authority and all parties generating identifiers. Having generator-specific markings reduces the burden on both the central authority and on identifier generators. As such, having entities establishing identifier formats include a space for these markings in identifiers is beneficial.

Clear division of the software space among generators: The need to divide the software space into non-overlapping segments for identifier generators remains necessary to avoid overidentification of software. However, the presence of the central authority allows a more deliberate association between software and identifier generators. In theory, the central authority could assign all such associations, but this would force the authority to track the entire software space and would mean that identities could only be generated in response to the authority's assignments. A less onerous path would be for entities establishing identifier formats to build in a natural association between software and the parties responsible for generating its identifiers, as was necessary with the unmanaged model in section 2.2.

The existence of the central authority also allows deliberate assignment of responsibility for identifier generation to address important coverage gaps through use of "third party" identifier generators, i.e., parties that are not software distributors but still have an interest in the completeness of the identifier space. Such parties might include inventory and vulnerability tool publishers who want their tools' results to support other data set correlation. The curating role of the authority allows such parties to contribute to the identifier space without increasing the risk of overidentification.

Push identifiers with software: The central authority's searchable database of identifiers provides an adequate means by which any identifiers might be looked-up by users. As such, pushing identifiers with software is not required for this model to work. However, pushing identifiers with software makes it easier to accurately use identifiers, both because it means users do not need to take the extra step of looking for identifiers from an external database and because it eliminates potential errors that this extra step could introduce. As such, when possible, distributors should push out software identifiers with software, although doing so is not strictly necessary for this model.

Minimize required information in identifiers: This is a requirement for this model for the same reason it is required under the model in section 2.2—reducing required information in identifiers reduces the cost and complexity associated with identifier creation and thus reduces disincentives to potential identifier generators. The presence of a central authority does not change the need for a large community of identifier creators, so entities establishing identifier formats need to ensure the identifier format does not create unnecessary burdens on identifier generators.

Incentivize identifier creation: This model requires identifiers to provide sufficient coverage of the software space so incentivizing identifier creation remains a priority. The same mechanisms to incentivize creation in section 2.2 also work here. Because of the presence of a central authority, there is also the opportunity for targeted filling of gaps in the identifier space. For example, the central authority could assign some party responsibility to create identifiers for pieces of software not otherwise covered, including legacy software that no longer has a natural identifier "owner."

Ensure long-term operation of the central authority: The software identifier ecosystem defined in this path will be viable only if the central authority remains in operation. Therefore, it is critical that long-term support for this authority is secure and well-resourced to keep it from becoming a bottleneck for the whole ecosystem. In support of this, search mechanisms associated with the central database will need to be well-defined and supported by records that reliably record the associations between software identifiers and software.

2.3.2 Expected Path Outcomes

As with the model described in section 2.2, this model relies upon the creation of a large cadre of identifier generators creating a critical mass of software identifiers to cover the software in enterprises. While the central authority could generate some identifiers itself, this is not its primary role, and the success of the model remains dependent on having multiple active identifier generators.

It is difficult to predict whether the managed or unmanaged models would lead to better coverage of the software space. The managed model might have fewer software distributors generating identifiers due to the added steps for coordinating with the authority, but it also can more easily support third-party identifier generators. Whether the latter would make up for the former is currently unknowable.

One significant advantage of the managed model over the unmanaged model is the opportunity to improve the overall quality of the identifier space. Any model that involves the maintenance of a repository of identifiers (be this repository centralized or distributed) will offer an authority the opportunity to review the entries in the repository and identify issues such as inconsistent embedded information, overidentification, and granularity errors. The authority will also be able to identify important coverage gaps in the identifier space by monitoring for missed queries against its identifier repository. As a result, assuming the central authority has sufficient resources, this model offers an identifier ecosystem that is more robust and more responsive to user needs than the unmanaged model.

Cost is one significant challenge of the model described here. The central authority will need funding to operate and its ability to support the community will be commensurate with the resources it is given. Moreover, the identifier ecosystem will only last as long as the central authority remains active, so long-term funding must be guaranteed.

Likewise, this model requires extra steps by identifier generators as they will need to submit created identifiers to the central authority for inclusion in the central repository. Having all identifiers submitted to the repository is the only way this repository will ever be complete, as it is impractical for the central authority to try to find and collect all identifiers produced across the global software ecosystem on its own. While automation could make publication to the central authority a trivial task, it might still deter some parties from supporting the identifier ecosystem, especially if the authority is affiliated with specific national, government, or commercial entities who might be viewed unfavorably in some circles.

Finally, as with the unmanaged model, the model described in this section will never be able to completely cover the global software space. While the central authority allows this model to be more responsive when critical coverage gaps are observed, it will generally not be able to prevent those gaps from occurring in the first place, at least for a time. The supplemental mechanism for characterizing identifiers described in Section 2.5 thus remains relevant under this model.

Figure 3 shows the process of identifier creation and use for managed, defined identifiers. Unlike the unmanaged, defined identifier model, the identifier generator is not necessarily the source of the identifier user's software. Because of this, an identifier authority serves as a middle party, receiving identifiers and associated software properties from the identifier generator and then serving queries from the identifier user to return the appropriate software identifier.

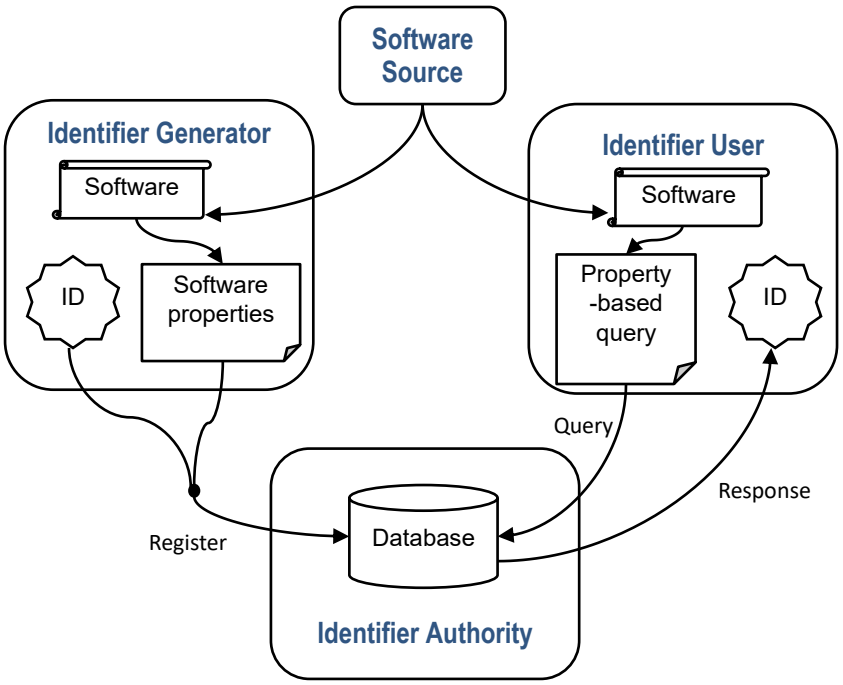


Figure 3: Managed Defined Identifier Creation and Use

2.4 Path 4: Intermediate Models for Defined Identifiers

While Sections 2.2 and 2.3 define two models based on whether there is a requirement for identifier generators to coordinate externally, there are actually many variants between the described models of "no external coordination" and "a single, central authority." Possible alternatives include varying responsibilities taken on by an authority or the possibility of a "distributed" authority with federated nodes. From some perspectives, these variations make little difference—anything more than a completely unmanaged model will require activities by identifier generators to coordinate with outside parties, which may be undesirable in any form. However, variations in the active management processes can have a significant impact on factors such as the level of resources needed to manage the ecosystem, the authority's ability to manage the quality and coverage of identifiers, and the degree of trust stakeholders have in the authority and ecosystem.

Ultimately, the key decision will be whether the community chooses to pursue an unmanaged model or one that requires active coordination by identifier generators. If the latter is chosen, the community will need to figure out what design for active management best fits their needs and resources. The path of having a central authority is one possible option among many.

2.5 Path 5: Unidentified Software Descriptor to Augment Paths 2, 3, and 4

None of the paths for defined software identifiers can cover the full global software space. Having every party that finds an unidentified piece of software invent an identifier for it is a detriment because it is not only unlikely that these ad hoc identifiers will correlate with anything, but it will likely lead to overidentification of the software in question. However, simply ignoring software that lacks an identifier will leave significant visibility gaps in any IT management activity and thus is not a viable option either. A good solution for filling gaps in a defined software identifier space takes on the features of a good solution to inherent identifiers (Section 2.1) within a more limited scope.

2.5.1 Actions to Pursue this Path

Standardize a structure to characterize unknown software: The data structure should allow one to capture inherent (e.g., size, hash) and asserted (e.g., software name, version) characteristics of a piece of software. These structures can then be used as a label for data artifacts in lieu of a software identifier. Unlike a software identifier, which requires different parties provide exactly the same identifier for correlation, a descriptive structure does not depend upon exact matches and correlation could instead be done using approximate string matching and record linkage over the set of provided characteristics. While more labor-intensive and less precise than identifier-to-identifier correlation, given enough descriptive elements in a structure, it should be possible to enable high-confidence correlations using these descriptive structures.

2.5.2 Expected Path Outcomes

If this augmentation to a defined identifier solution is adopted, then parties unable to find a needed identifier have a fallback method for indicating the given software in data artifacts. While any defined identifier solution should minimize gaps, a descriptive structure that supports correlation within the inevitable coverage gaps is still important. One expectation is that this fallback option should be used to identify gaps and inform community efforts to establish processes to reduce the gaps.

2.6 Path 6: More Than One Software Identifier Format

The ultimate goal of the paths Section 2 describes is to facilitate correlation by ensuring identifiers exist and are available where needed. The above paths were described as if there were only one identifier format in use, but this is not necessarily the case: a successful identifier ecosystem can also be created using multiple identifier formats, provided all formats meet the key requirements outlined in this document in terms of both their availability (section 2) and granularity (3).

The key consideration for a multi-identifier format ecosystem is the avoidance of overidentification between formats. As noted in the discussion of defined identifiers, overidentification occurs when a single piece of software is associated with multiple identifiers, and it is avoided by clearly dividing the global software space into disjoint sets for different identifier creators.

Division of the namespace to combat overidentification can be complicated if inherent identifiers are involved. The entities establishing and maintaining identifier formats for the respective inherent identifiers might define the scopes of their identifiers in a way to avoid overlap, but it would likely still be technically possible to generate an identifier for software outside the identifier's intended scope. Since any party in the world can generate inherent identifiers and use them to label data artifacts, it becomes likely that multiple inherent identifiers would be generated and used for a single piece of software. As a result, overidentification is more likely to occur in a multiple identifier format ecosystem when one or more identifier formats are inherent. For this reason, it will likely be preferable to coalesce the software identifier ecosystem around a single inherent identifier if an inherent identifier that meets all other user requirements is developed.

By contrast, an ecosystem with multiple defined identifier formats creates no additional challenges. This is because defined identifier formats already need to include mechanisms to prevent overidentification and these same methods can be directly applied to managing a multiple identifier format ecosystem. The division of the global software space between different formats does not need to be explicitly defined. As long as each party creating identifiers is only doing so using a single format, then overidentification will be avoided. Multiple defined identifier formats do add some complexity to an ecosystem. For example, if multiple central authorities exist, they will need to coordinate with each other to avoid overlaps in coverage, and users might end up needing to query multiple identifier repositories to find the identifier they need. Likewise, in unmanaged ecosystems, the way in which the different formats assign responsibility for identifier generation would at least need to be compatible, lest they assign identifier creation responsibility to two distinct parties for the same piece of software. However, having multiple defined identifier formats also potentially increases the number of parties generating identifiers overall, which could lead to greater coverage of the global software space, so there could be a benefit to such an ecosystem as well.

A successful software identifier ecosystem can be created around one or around many identifier formats, provided that overidentification is managed and all formats meet the availability and granularity requirements. Identifier formats that do not meet the requirements for availability and granularity do not usefully contribute towards an effective software identification ecosystem, whether they are used alone or in coordination with other formats. As there are currently no existing formats that adequately meet availability and granularity requirements, developing at least one format that meets ecosystem requirements remains the priority.

3 KEY REQUIREMENT: SUPPORT GRANULARITY OF DATA ARTIFACTS

The existence of identifiers and their availability is clearly necessary for any activity that depends on those identifiers. However, it is also critical that the software identifiers be able to support the activities and use cases that motivate the correlation efforts in the first place; that is, when correlation is performed over correctly tagged data, the results need to be usable and useful.

The value of correlation results is reduced when successful correlation (i.e., matches between correctly labeled artifacts) nonetheless leads to misleading results. This can happen when there is a mismatch between the granularity of the identifier and the granularity of the information to which it is attached. For example, consider an identifier that corresponds to multiple variants of a piece of software, such as a software identifier corresponding only to major and minor releases, where all "update" releases within a minor release are lumped together under the same identifier. Assume one of those variants (but not others) has a software vulnerability that is captured in a vulnerability report annotated with the corresponding software identifier. At the same time, assume a different variant of the same piece of software is found in a software inventory scan, the results of which are likewise annotated with the corresponding identifier. Should these two datasets be correlated, there would be a match between the inventory and vulnerability report, because the same identifier appears in both. However, this is a false positive match, because both artifacts refer to different variants of a piece of software and the variant found in the inventory is not the variant with the vulnerability. In this example, the identifier design itself (namely the decision to have a single identifier correspond to multiple software variants) created a situation where two correctly annotated data artifacts nonetheless led to an incorrect correlation.

Problems can also occur when identifiers are finer grained than the artifacts to which they are attached. Data artifacts can sometimes refer to groups and ranges of software variants. A common example of this are vulnerability reports that apply to a range of versions of a piece of software. Some vulnerability reports can refer to dozens or hundreds of distinct releases of a piece of software. In such cases it can be difficult to know the complete list of distinct software variants in a given range, much less find identifiers for them. However, failure to cover the full range of software variants in a data artifact with corresponding identifiers risks false-negative correlations when a piece of software is within the range of associated software, but its specific software identifier is not included in the artifact's list of associated software.

For a software identifier scheme to support reliable correlation results, the scheme must support annotation of data artifacts with a range of granularities. Of course, much of the onus remains on the party assigning software identifiers to these data artifacts and their need to make sure they pick the correct identifiers that match the artifact's subjects. However, as shown above, design decisions in the software identifier scheme can create issues that make it difficult or even impossible to avoid incorrect correlations even when the optimal identifier(s) are chosen.

The paths below focus on how a software identifier scheme would be able to support the full range of granularities needed to support data artifacts associated with key cybersecurity use cases. Both paths require identifiers that can correspond to specific software variants with fine granularity to avoid the problem with false-positive correlations described above. The two paths differ in how they handle mapping identifiers to

artifacts that cover a range of software variants, with one path defining a standard expression for ranges and another using a central database to ensure complete enumeration of identifiers within of a given range.

3.1 Path 1: Fine-granularity Identifiers with Grouping Expressions

In this model, identifiers all refer to a single variant of a given piece of software (that is, they are "fine-grained"). This design choice is necessary to avoid situations in which multiple distinct software variants are represented by the same software identifier, leading to potential false-positive matches.

To handle situations where data artifacts cover a range of software variants, this model employs a grouping expression. A grouping expression represents a range of software variants and can be mechanically compared against individual identifiers or other grouping expressions to determine whether there is overlap. A grouping expression might be built into the identifier format itself or could be a separate syntactic construction against which identifiers can be compared. Grouping expressions can be used to annotate data artifacts and allow ranges and other sets of variations of software to be expressed without the need to fully enumerate the associated identifiers.

The CPE identifier standard provides an example of grouping expressions. CPE version 2.3 identifiers consist of eleven fields, each corresponding to a particular property of a software product (e.g., its version, edition, language, hardware architecture, etc.). By using asterisks in one or more of these fields, the identifier is used to represent all software variants that share the same non-asterisk properties. A further CPE applicability language allows other types of groups to be defined, such as expressing sets of software variants that exist between two versions.

The main challenge with grouping expressions is that, to allow mechanical comparison of a software identifier and a grouping expression, it is necessary for the software identifier to embed information regarding the associated software. Specifically, any property of the software to be used as a constraint in defining a group must be captured in the software identifier itself. For example, to evaluate an identifier against an expression that represents a range of software versions, each identifier would need to incorporate the version of the software it represents. The embedded information need not be human readable (although there are advantages if it is), but the information must be present. Moreover, the format needs to allow the information to be mechanically extracted, such as by embedding information in well-defined locations.

As observed in both defined identifier sub-paths in sections 2.2 and 2.3, embedding information in software identifier can increase the amount of effort being asked of identifier generators and creates the opportunity for errors. For these reasons, both the defined identifier sub-paths recommend minimizing the amount of information embedded in identifiers. At the same time, the type of information generally needed to differentiate software variations in data artifacts (e.g., version, update, edition, supporting software framework) generally cannot be mechanically extracted from the executable file of a piece of software, such as personnel would use to generate the inherent identifiers described in section 2.1. As such, there is a tension between the needs of paths associated with ensuring identifier availability and the requirements needed for grouping expressions. Should this model be selected, a tradeoff between these competing requirements will need to be identified.

In addition to entities establishing and maintaining identifier formats supporting the embedding of information within identifiers, the process of software identifier creation needs to be managed to ensure this embedded information is normalized. Normalization failures can lead to correlation failures when comparing identifiers to grouping expressions. For example, consider the case where different identifiers for plug-ins for the Firefox web browser indicate their software environment using one of "Firefox", "Firefox Browser", or "Fire Fox". A grouping expression for all Firefox browser plug-ins would generally only use one of these terms; it will lead to correlation failures against identifiers that use a different term. The data normalization challenge is significant as can be seen in the CPE dictionary, which has many examples of slightly different terms for the same thing. To ensure

that correlation occurs successfully, it is necessary to have mechanisms that lead to normalization of the embedded data so that identifier creators use the same terms for the same thing.

Of existing software identifier standards, only CPE currently supports grouping expressions since it provides a well-defined structure for embedding key software properties. However, CPE continues to struggle with challenges around normalization of embedded information. Other identifiers either do not embed information (SWID tag identifiers and OmniBORs) or the information they embed varies based on certain factors (purls), and thus do not currently support grouping expressions.

3.1.1 Actions to Pursue this Path

Identifier formats support embedded information: To support grouping expressions, software identifier format owners need to ensure that their identifier format supports embedding key properties that are frequently used for grouping. These properties must be required in all generated identifiers and would need to be embedded in a way that allowed mechanical extraction. (For the purposes of this section, the party "generating" the identifier might be a tool creating an inherent identifier or some assigned party generating a defined identifier.) As discussed earlier, to minimize burdens on identifier generators, required embedded properties should be kept to a minimum. NIST's National Vulnerability Database (NVD) provides useful guidance for this, as it provides a large set of software vulnerability data artifacts mapped to CPE grouping expressions. Analysis of the properties used in these expressions over the last three years reveals that capturing software name and vendor, software version, software update level, the target software environment (e.g., host OS) would meet the needs of over 95% of grouping expressions needed to identify software impacted by vulnerabilities. Adding the software edition increases coverage to over 99% and then adding the target hardware environment (e.g., X86 architecture) achieves over 99.97% coverage.

The selected properties would need to be required in all identifiers to provide the described level of grouping support. For software products where the given property is not relevant (e.g., software that does not have multiple editions), entities establishing identifier formats would need to define a way for identifier generators to explicitly denote that the property was not applicable within the identifier.

Identifier formats allow separate identifiers for any software variation: While only the selected properties should be mandatory, it is also necessary for the format to allow identifier generators to further distinguish between software variants. For example, if a particular piece of software had separate variants for English and Spanish-speaking countries that otherwise shared all other properties (i.e., name, version, update, etc.) the generator would need to be able to assign these variants separate identifiers, lest two software variants be mapped to a single identifier and run the risk of false-positive correlations. As such, entities establishing identifier formats need to allow disambiguation based on properties other than the mandatory ones.

Development of a grouping expression: In addition to creating the necessary structures in the software identifier format, entities establishing identifier formats need to develop the syntax and mechanisms for the grouping expressions. Grouping expressions might be based on the identifier format itself or could be an entirely separate syntactic construct. (CPE employs both approaches in parallel.) The same organization should oversee both the identifier format and the grouping expression syntax, since they will be interdependent. The syntax of the grouping expression may constrain the types of groups that can be readily defined, so it will need development with an understanding of the types that groups commonly needed in data artifacts. NIST's NVD can be a useful resource for this analysis.

Support normalization of key properties: The other key challenge that needs to be addressed is normalization of embedded data. As noted earlier, normalization failures can lead to correlation failures. Entities establishing identifier formats will need to provide clear guidance on how to populate certain properties. Guidance can include things like whether capitalization is significant, the preferred authoritative source of software names and editions, and similar guidance. Another way of improving consistency is the creation of validated tools for identifier generation. Such tools will provide greater consistency in identifier generation. Finally, oversight of

the identifier space can detect and help mitigate the use of inconsistent terms. Even for paths that do not define a central authority for the software identifier scheme, some party (such as a volunteer community support group) could still provide a path to identify and reconcile inconsistencies that arise within the namespace.

3.1.2 Expected Path Outcomes

If this path is followed, software data artifact authors will be able to annotate these artifacts at the necessary granularity, either using a single identifier to represent a specific software variant or using a grouping expression to represent a range of variants. A key benefit of grouping expressions is that it allows correlation computing between an identifier and a grouping expression, or between two grouping expressions directly without needing any external information. Likewise, it should be possible for any party to create a grouping statement that aligns with their data artifacts.

The primary challenge to this path's success will be the degree to which collecting and embedding information complicates identifier creation. For defined identifiers, this increases the burden on identifier generators, although hopefully automation and tools will reduce this burden in most cases. The situation is even more fraught for inherent identifier schemes since key properties (i.e., software name, version, edition, and update) are not "inherent" to a binary file and thus could prove difficult to extract consistently. Personnel attempted to extract these properties from binary files in multiple efforts in support of CPE identifiers, but no mechanism ever arose that created the necessary consistency across different identifier creators. As such, combining this path with the inherent identifier path remains an unsolved challenge.

3.2 Path 2: Fine-granularity Identifiers with Support for Complete Identifier Enumeration

In this path, identifiers are built to express a fine level of granularity, as was the case for the grouping expression path. This allows identifiers to distinguish between software variations in artifacts and minimizes false-positive correlations.

To support expressing ranges of software variants, the ecosystem supports computation of completely enumerated sets of software variants within a given range. This allows artifact authors to ensure that no software variants are missed when expressing software groups. Since artifact authors are likely to only have a small number of software instances available to them when creating artifacts, fully enumerating the software variants within a range requires access to a searchable database that holds a complete list of all identifiers. Artifact authors would query the database using the bounds of the group they wish to define. The database would then provide a complete list of all software identifiers that exist within the bounds of the group.

Data artifacts could either be annotated with a list of identifiers retrieved from the database or with the query parameters that describe the group. The latter offers some advantages as it will often be more concise than a complete set of identifiers. Query-based group description also enables a practice which is both in use and somewhat dangerous: defining open groups (for example, "version 1.2.3 and later") where new members of the group might be added later. Open groups are dangerous because the claim assigns a property to versions (for example) 1.2.4 and 2.1.0 which may not actually hold when that future version of the software is in fact released. Capturing the query parameters provides a capability similar to the grouping expressions described in section 3.1, with the key difference that correlation between an identifier and the grouping expression cannot be computed without the use of the central database.

The key challenge of this path is establishing a complete, searchable database of software identifiers. The database's completeness is crucial because any missing identifiers will create gaps that can lead to correlation failures. Such a database is already part of the path for a managed, distributed model for defined identifiers (section 2.3), and thus does not add significant new challenges to that model. For the other paths around identifier availability, creating such a database represents a daunting challenge. The distinguishing feature of

the unmanaged, distributed model (section 2.2) is that it does not require identifier generators to coordinate with any central authority, and having generators push identifiers to a central database effectively switches to the managed model.

The only way to have an unmanaged model with a central database of identifiers would be for the database owners to scrape the entire global software space to collect published identifiers. This would also be necessary for the inherent identifier path (section 2.1). It is difficult to imagine that the required completeness of the database could be achieved by one party scraping the global software space for identifiers. For this reason, the use of a central database to support complete enumeration of software identifiers is likely only practical in conjunction with the path for managed, distributed software identifier generation.

3.2.1 Actions to Pursue this Path

Assuming that the managed, distributed model for identifier generation is employed, the actions needed to implement this solution are relatively minor.

Identifier formats allow separate identifiers for any software variation: As with the path for grouping expressions, entities establishing and maintaining identifier formats will need to ensure that identifier generators can assign different identifiers whenever they observe different software variants, regardless of the characteristic that differentiates those variants. Since this path likely does not involve formats with specific fields dedicated to certain software properties, format authors will likely need to provide explicit guidance regarding factors distinguishing variants, while emphasizing that any list of factors is incomplete and that there may be other reasons to provide separate identifiers for different releases of software.

Develop a query language for software ranges: The party that operates the central database of identifiers will need to develop a mechanism not only to return a single identifier given information available from possession of a piece of software (as is needed under the managed, distributed identifier creation model in section 2.3) but would also need to develop a way to query for all identifiers within a given range of software variants. This design requirement relies on the same understanding of commonly used software groups as the development of a grouping expression (Section 3.1). Moreover, it will be advantageous for this group to develop a way to capture this query as text so it can be embedded in data artifacts.

Ensure long-term operation of the central database: The software identifier ecosystem defined in this path will only be viable if the central database remains in operation. Therefore, it is critical that long-term support for this database is secure. In support of this, search mechanisms associated with the central database will need to be well-defined and supported by records that reliably record the associations between software identifiers and software. Specifically, these records will need to record all associated software properties that could be used to bound groups.

3.2.2 Expected Path Outcomes

In this model, data artifacts can be annotated with both singular and groups of identifiers reflecting the scope of the associated software. The key difference is that correlating against a group annotation requires interaction with the central database at some point, either when the artifact is annotated (if the artifact author fully enumerates the relevant identifiers in the artifact) or at the time of correlation (if the artifact uses a query expression). Because of this, ensuring the long-term availability of the database is a key requirement for this model.

The main advantage of this model is that it does not require any information to be embedded in software identifiers. The format of the identifiers themselves becomes largely irrelevant in this model as it is the central database that is the enabler of correlation rather than the identifiers themselves. However, this will not reduce the overall burden on identifier generators since they will still need to collect all the software properties that would be used to bound a group and provide this information to the central database along with the generated identifier. This is needed so the central database can correctly respond to queries for group inclusion. Identifier

generators would also need to ensure this information was reasonably normalized, although ingest capabilities in the central repository could help with quality control.

4 INTERACTIONS BETWEEN PATHS FOR THE TWO KEY REQUIREMENTS

Key requirement 1 (Section 2) and Key requirement 2 (Section 3) are interrelated in complex ways. Therefore, the paths that might accomplish each requirement are also interrelated in complex ways. Just as the paths for meeting each key requirement may support or trouble the others, the paths between key requirements may present solutions or challenges for each other.

For example, inherent identifiers (Section 2.1) pairs much more naturally with identifier enumeration (Section 3.2) than identifier group expressions (Section 3.1). They pair better because inherent identifiers have a very hard time embedding human-conceptualized semantic information such as the order in which the software was produced, which has a lot to do with sequential version numbering.

If identifier grouping expressions are used (Section 3.1) with any of the defined identifier options, then part of the identifier format that is agreed upon must define the grouping expression syntax and semantics. If different software identifier formats are in use in the ecosystem (Section 2.6), the different formats may need to agree on a single identifier grouping syntax and semantics, or at least agree on how to translate the version grouping expressions accurately between formats. If identifier enumeration is employed, then multiple identifier formats are slightly easier to accommodate, because there are no grouping functions to worry about

5 CONCLUSION

This paper characterizes the requirements and activities necessary to establish a harmonized software identification ecosystem to facilitate greater automation, inventory visibility, and the multifaceted value proposition of SBOM's broad adoption.

There are two key requirements for an effective software identification ecosystem:

1) Availability of software identifiers (that is, labels for specific versions of software items, such as an applications, libraries, etc. which conform to a defined format).
2) Software identifiers that support both precise (i.e., a single software variant) and broad (that is, a large range of software variants).

Any solution that productively contributes to a more robust software identifier ecosystem needs to address both requirements. Some existing software identifiers do a good job of addressing one of these two requirements: OmniBOR could be used to provide a unique and immediately available identifier for most of the world's software, while CPE supports grouping expressions (i.e., a range of software variants that can be compared by name, version, architecture, language, etc.) to meet just about any granularity need for data artifacts. However, no existing software identifier today adequately meets both requirements for a sufficient percentage of the software used in modern enterprises.

In the presentation of the potential paths forward, a few key themes emerge about the operational entities that may participate in a software identifier ecosystem:

- If defined identifiers (i.e., identifiers created only by certain parties and only at one point in time) are used, identifier generators (i.e., organizations who create identifiers) will need to capture certain common properties of software (e.g., name, version, architecture). This information is necessary to support data artifacts that apply to groups of software, such as CVE Records.
- A central authority that establishes common rules, assigns responsibilities, and identifies and addresses subsequent issues for the identifier generators would likely improve the overall accuracy

and robustness of the software identifier ecosystem. Such an authority would need to be securely funded to ensure its longevity and keep it from becoming a bottleneck.

- Without a central authority, the quality of identifiers (and thus the quality of the correlations supported) will depend on the individual efforts of the identifier generator community.
- Only adoption of inherent identifiers (i.e., those that can be generated by any party at any time via the artifact itself) will remove the need for a large community of identifier generators. Inherent identifiers provide optimal availability, but marrying inherent identifier schemes with the non-inherent software properties required to support grouping remains an unsolved research challenge.

Three potential options that build off existing software naming schemes are described here for community discussion. While they do not explicitly address the operational entities (e.g., central authority, identifier generators) that may participate in a software identifier ecosystem, the options can serve as starting points to refine the merits of various operational models:

- If the community wishes to pursue **Inherent Identifiers**, then **OmniBOR** provides a strong starting point. The biggest challenge will be around fitting OmniBOR identifiers to the varying granularities of software data artifacts that must be annotated. One option would be to research how non-inherent properties (such as software name, vendor, and version) could be mechanically collected and used to provide the properties needed for grouping expressions. There may be ways to mechanically collect this information from systems (e.g., from software package or installation managers, certificates associated with signed code, filesystem information). Alternately, if the community wishes to pursue a complete repository of identifiers, the National Software Reference Library (NSRL) provides a template for such an endeavor, although such an effort is likely not scalable.
- If the community wishes to pursue **Defined Identifiers with Grouping Expressions**, then **CPE's** existing applicability language already provides a robust grouping expression capability. CPE's challenge would be evolving to a more distributed production model to address scalability and sustainability challenges. Alternately (or concurrently), **purl** could be the basis of a solution. purl already supports a distributed creation model but would need to be expanded to cover software beyond those distributed in packages and to embed necessary information. On the latter point, some purl formats already embed key information, like software name and version, but to fully support grouping expressions, this would be required of all purls in a way that allows easy mechanical extraction.
- If the community wishes to pursue **Defined Identifiers with More Comprehensive Enumeration Support (e.g., searchable database)**, then **purl** is a promising starting point due to its existing mechanisms for distributed identifier generation. The scope of purl's supported software would need to be broadened beyond just packaged software and the tools that generate purls would need to add a step to register all identifiers with a central authority. Likewise, all parties would require reliable access to the database needed to map identifiers against range queries.

Appendix A

Related work

The National Telecommunications and Information Administration published Software Identification

Challenges and Guidance on March 30, 2021 and Software Identity Discussion and Guidance on July 6, 2020. These documents inform and provide additional background on the challenges and state of software identification.

Software identification is not a widely studied academic topic. To assess whether a high-quality systematic review of the topic has already been conducted, we searched the proceedings of ACM Computing Surveys, the top academic venue for publishing such reviews. The search for related surveys was executed on May 10, 2023, via searches on Google Scholar. The results are shown in Table 1. Titles returned in searches were assessed for plausible connection with the relevant senses of "software identification" and potentially relevant papers are referenced under the search, once the number of search results was reduced to a number that could be perused manually.

Table 1: Search Terms for Systematic Literature Reviews on Software Identification

Search term	Number of results
"software identifier" source:"CSUR"	0
"software naming" source:"CSUR"	0
"software identification" source:"CSUR"	0
"package URL" source:"CSUR"	0
"software bill of materials" source:"CSUR"	0
"common platform enumeration" source:"CSUR" • Figueroa-Lorenzo S, Añorga J, Arrizabalaga S. A survey of IIoT protocols: A measure of vulnerability risk analysis based on CVSS. ACM Computing Surveys (CSUR). 2020 Apr 26;53(2):1-53. • Hamdani SW, Abbas H, Janjua AR, Shahid WB, Amjad MF, Malik J, Murtaza MH, Atiquzzaman M, Khan AW. Cybersecurity Standards in the Context of Operating System: Practical Aspects, Analysis, and Comparisons. ACM Computing Surveys (CSUR). 2021 May 8;54(3):1-36.	2
software identifier source:"CSUR"	708
"software" "identifier" source:"CSUR"	328
"software" "identifier" "identification" source:"CSUR"	198
"software" "identifier" "identification" source:"CSUR" since 2000 • Li G, Liu H, Nyamawe AS. A survey on renamings of software entities. ACM Computing Surveys (CSUR). 2020 Apr 16;53(2):1-38. • Schrittwieser S, Katzenbeisser S, Kinder J, Merzdovnik G, Weippl E. Protecting software through obfuscation: Can it keep pace with progress in code analysis?. ACM Computing Surveys (CSUR). 2016 Apr 5;49(1):1-37. • Bernardi S, Merseguer J, Petriu DC. Dependability modeling and analysis of software systems specified with UML. ACM Computing Surveys (CSUR). 2012 Dec 7;45(1):1-48. • Figueroa et al (same as above)	164

• **Qasem A, Shirani P, Debbabi M, Wang L, Lebel B, Agba BL. Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. ACM Computing Surveys (CSUR). 2021 Mar 5;54(2):1-42.** • **By page 4 of the results, it was clear that the papers were about using software to identify other things (proteins, biometrics, energy consumption, facial recognition, spam filtering etc.) and were off topic. Manual review stopped after 70 titles has been assessed.**	

The next step is to read the abstracts of the six survey papers surfaced by search summarized in Table 1 and assess them for relevance. The relevance criteria are whether the paper is about (1) identification of software entities and (2) a software engineering lifecycle.

- Figueroa et. al is about assessing the risk to an industrial environment; specifically, risks posed by vulnerabilities identified in software. Asset management and software identification is not a focus, so this paper is assessed not relevant.
- Hamdani et. al is about understanding various cybersecurity standards and tools for assessing compliance with those standards. I assess this paper to be not relevant.
- Li et. al is about whether variable names and other semantic items within source code have useful names that identify their purpose to a software engineer. Li et al also discusses benefits of good name selection during software development and automatic provision of semantically useful variable identifiers. While this is about software engineering, it is not about software identification at the right level of granularity. Therefore, I assess this paper is not relevant.
- Schrittwieser et. al is about whether the functionality of intentionally obfuscated software can be recovered via reverse engineering. This is about function identification, rather than software identification, and so I assess it to not be relevant.
- Bernardi et. al is about using Universal Markup Language (UML) as an abstraction layer for software engineering. The topic focuses on approaches to UML modeling and what parts of the software development lifecycle are supported by what kinds of UML tools. How software elements are identified does not appear to be a topic; therefore I assess this paper to not be relevant.
- Qasem et al is about vulnerability detection techniques used on firmware images. Unique identification of the firmware is not a topic of focus, so I assess this paper to be not relevant.

None of the six potentially relevant surveys pass the assessment of their abstracts for relevance. This lack of academic survey influences our methodology to focus more on practitioner knowledge and embedded expertise. This search does not indicate there is no academic work on software identification; however, it suggests there is no established academic community focused on the topic. Collecting and synthesizing disparate academic work on the topic into such a community is not our goal. We will focus on collecting what is known within the practitioner community.

There is academic work relevant to naming software. Some of this lies within the area of IT asset management. However, guides such as NIST's SP1800-5 tend to assume the tools are handling consistent names for software elements, rather than discussing how to make such names consistent. Our focus is exactly this discussion of the community processes necessary to make names consistent and reliable across the whole cybersecurity community. Asset management, at best, focuses on consistent naming within a single organization.

Software identification is about naming and how we use language. There is a well-developed philosophy of language literature on the topic of names and naming, for example. However, besides confirming that naming and identification is a social task without *a priori* correct answers—which the software identification already knows as they have been hard at work trying to agree at the social task—this literature does not offer much as far as practical advice.

There is a body of work on software identifiers, including numerous identifiers that are in use by various communities today. The observations and potential paths outlined in this paper have pulled heavily from the lessons learned from these prior efforts. We do not provide a detailed listing and study of existing software identification efforts. Some concepts that might be notable by their absence—such as semantic versioning—could be absent because of their ubiquity or because they are not exactly located within software identification. Take semantic versioning as an example: it is a scheme to denote how versioning works within a software project, whereas the software identification work focuses more on distinguishing software projects from each other. Once a software project is readily and reliably identifiable, it is comparatively straightforward to understand how the project does versioning.