The State of Software Diversity in the Software Supply Chain of Ethereum Clients

Noak Jönsson

***Abstract*—The software supply chain constitutes all the resources needed to produce a software product. A large part of this is the use of open-source software packages. Although the use of open-source software makes it easier for vast numbers of developers to create new products, they all become susceptible to the same bugs or malicious code introduced in components outside of their control. Ethereum is a vast open-source blockchain network that aims to replace several functionalities provided by centralized institutions. Several software clients are independently developed in different programming languages to maintain the stability and security of this decentralized model. In this report, the software supply chains of the most popular Ethereum clients are cataloged and analyzed. The dependency graphs of Ethereum clients developed in Go, Rust, and Java, are studied. These client are Geth, Prysm, OpenEthereum, Lighthouse, Besu, and Teku. To do so, their dependency graphs are transformed into a unified format. Quantitative metrics are used to depict the software supply chain of the blockchain. The results show a clear difference in the size of the software supply chain required for the execution layer and consensus layer of Ethereum. Varying degrees of software diversity are present in the studied ecosystem. For the Go clients, 97% of Geth dependencies also in the supply chain of Prysm. The Java clients Besu and Teku share 69% and 60% of their dependencies respectively. The Rust clients showing a much more notable amount of diversity, with only 43% and 35% of OpenEthereum and Lighthouse respective dependencies being shared.**

***Sammanfattning*—Mjukvaruleverantörskedjan sammanfattar all resurer som behövs för att producera en mjukvaruprodukt. En stor del av detta är användningen av öppen källkod. Trots att användningen av öppen källkod tillåter snabb produktion av nya produkter, utsätter sig alla som använder den för potentiella buggar samt attacker som kan tillföras utanför deras kontrol. Ethereum är ett stort blockkedje nätverk baserad på öppen källkod som försöker konkurera med tjänster som tidigare endast erbjudits av centraliserade instutitioner. Det finns flera implementationer av mjukvaran som implementerar Ethereum som alla utvecklas oberoende av varandra i olika programmerings språk för att öka stabiliteten och säkerheten av den decentraliserade modellen. I denna rapport studeras mjukvaruleverantörskedjorna av de mest populära klienterna som implementerar Ethereum. Dessa utvecklas i programmeringspråken Go, Rust, och Java. Dom studerade klienterna är Geth, Prysm, OpenEthereum, Lighthouse, Besu, och Teku. För att genomföra studied transformeras klienternas mjukvaruleverantörskedjor till ett standardiserat format. Kvantitiva mått används för att beskriva dessa leverantörskedjor. Resultaten visar en stor skillnad i storlek av leverantörskedjor för olika lager i Ethereum. Det visas att det finns en varienda mångfald av mjukvara baserat på de språk som klienter är utvecklade med. Leverantörskedjorna av Go klienter sammanfaller i princip fullt, medan de av Java klienter sammanfaller med en stor majoritet, och de av Rust klienter visar på mest mångfald i mjukvarupaket.**

***Index Terms*—Software Supply Chain, Dependency Graphs, Open Source Software, Software Diversity, Ethereum, Blockchain**

Supervisors: Benoit Baudry, César Soto-Valero

TRITA number: TRITA-EECS-EX-2022:183

I. Introduction

The software supply chain is comprised of all resources, human and technological, required to produce a software product [1]. A significant component of this software supply chain are package managers, which are programming language-specific collections of open-source software packages. Package managers distribute open-source packages through online repositories and provide methods of collecting all dependencies required for a certain package. These package managers are often referred to as ecosystems, as the packages they constitute are interconnected through dependencies. The use of open-source packages from these software ecosystems is not only limited to use for other open-source projects. Instead, it has been shown that 85% of the source code for an average enterprise application is from open source packages [2]. Although the reuse of open-source software packages can allow faster development of new projects, all dependent projects become susceptible to the same bugs that emerge in a dependency, as well as malicious code injections.

Software supply chain attacks are one of the most prevalent methods used by malicious actors in order to compromise software, and a growing concern for both developers and policy makers [3]. One common method is *Typosquatting*, where malicious actors release software packages with names that are slight spelling variations of popular open-source packages, hoping to trick developers into including these infected packages [4]. Malicious actors may also try to infect existing packages by gaining access to the repository where the source code is hosted either by social engineering or by hacking the account of someone who already has access.

The Ethereum blockchain is a distributed software platform built entirely using open-source software. Through the use of smart contracts, digital assets can be exchanged between the parties involved without the need for a centralized governing institution. These smart contracts can be written to provide functionalities such as financial services, digital art trade, online games.

There are exist several Ethereum clients, developed in different languages. A vast network of nodes, computers which run these clients, all communicate each other in order to host the Ethereum Virtual Machine. The clients are split into to groups; the execution layer (Eth1), which is responsible for appending new transactions to the blockchain, and the

consensus layer (Eth2), which is responsible for making sure the added transactions are distributed among all the nodes.

In this paper, we study the software supply chain of three pairs of Ethereum clients. The analysis is narrowed down to focus specifically on the software diversity of open source dependencies and their suppliers. The studied clients are GoEthereum and Prysm, developed in Go; Besu and Teku, developed in Java; and OpenEthereum and Lighthouse, developed in Rust. These clients are chosen for two reasons: 1) they include the most popular clients in use, combined they total over 90% of all nodes in the Eth1 execution and Eth2 consensus layer currently in use; 2) the existence of a client written in a particular language in both Ethereum layers.

The analysis was conducted by download and build each client from the the source code. Outputting the dependency trees of each client using their native package manager. Reformatting this output into a uniform format and finally performing analysis on each tree individually, and comparing trees from the same ecosystem.

In summary, this paper makes the following contributions:

- The notion Unified Dependency Tree as a way to study the diversity in the software supply chain of distinct Ethereum clients.
- Novel metrics regarding Ethereum Clients including: total dependencies, unique direct dependencies, unique transitive dependencies, and unique suppliers.
- Insights into the distribution of suppliers within different ecosystems which validates past research.

II. Background

A. Software Supply Chain

Software supply chains are all the resources required to produce a software product. This includes human resources, such as developers, teams, and larger organizations. Technical procedures such as automatic testing and build processes [5]. Lastly, this also includes other software products such as standard libraries, tools, and third-party software packages. The focus of this paper will be on the software diversity in software supply chains with regard to third-party open-source software (OSS) packages and their suppliers.

The use of OSS by developers to create a new product is a cornerstone of modern development practices. In order to feasibly facilitate the reuse and distribution of OSS, developers often rely on package managers. Package managers are programming language-specific repositories of OSS packages [6]. Not only do they host the source code for OSS packages, but they also provide methods for downloading, updating, and building packages. Examples of package managers are Gradle and Maven for Java, PyPi for Python, and Cargo for Rust. Go, which is used to develop two of the clients studied in this report, does not utilize a package manager. Although Go does not have central repositories, the language does provide a tool for downloading and updating packages.

In order to utilize the functionality provided by an OSS package in a project, a developer must declare it as a dependency. Software dependencies are packages that are required by another package in order to function. Declaration of dependencies is accomplished through the use of a file in the root of the project directory.

Listing 1 shows an example of how dependencies are listed for a project developed in the language Rust, utilizing the cargo package manager. Common for all package managers is to list the name of the package, which is unique. Most, although not all, package managers also require a specific version of the dependency package to be declared. Any package referenced explicitly as a dependency in a project is defined as a direct dependency on said project. As direct dependencies themselves may have their own dependencies, they are also dependencies to the project. These dependencies are defined as transient dependencies.

```
[package]
description = "OpenEthereum"
name = "openethereum"
# NOTE Make sure to update util/version/Cargo.toml
    as well
version = "3.3.4"
license = "GPL-3.0"
authors = [
    "OpenEthereum developers",
    "Parity Technologies <admin@parity.io>"
]

[dependencies]
blooms-db = { path = "crates/db/blooms-db" }
log = "0.4"
rustc-hex = "1.0"
docopt = "1.0"
clap = "2"
term_size = "0.3"
textwrap = "0.9"
num_cpus = "1.2"
```

Listing 1. Example of dependency declaration in Rust using Cargo.toml file.

B. Software Supply Chain Attacks

Software supply chain attacks are directed attempts to inject malicious code into a software package in order to compromise any and all software packages which are dependent on the targeted package. In 2021 the EU Agency for Cybersecurity reported that 66% of cyber attacks target the software supply chain [7]. Decan et al. [6] analyzed the trends in seven different software ecosystems. They found that a majority of software packages are dependent on a minority of core packages. This highlights how a successful and well-targeted attack can affect the majority of a software ecosystem.

Software supply chain attacks targeting the dependency tree can be split into two categories; those infecting existing packages and those that create new packages containing malicious code [4]. When infecting an existing package, culprits often rely on existing known vulnerabilities in the code. Otherwise, they need access to the project, which can be achieved through social engineering, i.e., manipulating their way to get maintainer privileges for the project or by gaining the credentials of a person who is a maintainer of the project. When creating new packages containing malicious code, the culprit must still inject the package into some software supply chain. This is most commonly achieved through Typosquatting, which is when a package given a name that is a slight spelling variation from that of a popular package. For example a package could

be named 'bloons-db' instead of 'blooms-db', as seen in Listing 1. Other ways of injecting a malicious package include creating a Trojan Horse, where a package claims to provide some functionality but has a built-in backdoor mechanism to allow culprits to extract data from the end-users of the project.

C. Software Diversity

Software diversity is a concept with a broad scope in the study of software development [8], [9]. In general, it refers to the existence of multiple software components which are functionally similar, but implemented and created in separate ways. The aim of software diversity is to encourage fault tolerance, security, and reusability [10].

This paper deals mostly with the concept of design diversity and managed natural diversity. Design diversity refers to the practice of independently developing multiple software projects according to the same specification. Utilizing these projects simultaneously yields a more fault tolerant system due to the independence of failures among the diverse solutions-[10]. Managed natural diversity emerges as a result of development practices. As open source licenses give anyone the right to copy, modify, and redistribute an OSS packages, this practice has the potential to yield vast amounts of software diversity [11]. The opposite of software diversity is mono-culture, where a single software supplier, or a single package, is heavily reoccurring in a software supply chain. Mono-culture provides malicious actors a definite target from which a malicious code injection could have a tremendous impact.

D. Ethereum Ecosystem

Ethereum is an open-source decentralised software platform used for finance, digital art, and a host of web3 applications [12]. Based on blockchain technology, Ethereum functions by allowing users to share and trade digital assets through smart-contracts, which are recorded in a digital ledger. The contents of the digital ledger are maintained and agreed upon by a vast number of nodes, which are computers that support the Ethereum Virtual Machine (EVM).

The Ethereum Foundation promotes design diversity, in the form of client diversity [13]. There is no official implementation, rather there are several clients developed in different programming languages, as to increase software diversity by leveraging several ecosystems of OSS packages.

The function of the execution layer (Eth1) is to add new blocks of transactions to the shared state of the network. Eth1 uses a proof-of-work (PoW) mechanism in order to ensure that the new state is valid. When a transaction occurs, and is to be added to the blockchain, nodes running an Eth1 client compete against each other in completing a computationally heavy task. The first node to complete the task is allocated the block, and all other nodes with point to it as the correct state. The currently available Eth1 clients, the language they are developed in, and the percentage of nodes running them are shown in Table I.

The function of the consensus layer (Eth2) is to make sure that the updated state of a new block being added to the chain is distributed amongst all the nodes in the network. Eth2 uses proof-of-stake (PoS) validation. This consensus method is more energy efficient, as no computationally heavy task is required. In this method, nodes stake their own capital as collateral in order to ensure that they behave correctly. The currently available Eth2 clients, the language they are developed in, and the percentage of nodes running them are shown in Table II.

Tabell I
EXECUTION LAYER (ETH1) CLIENTS

Client	Programming Language	Distribution
GoEthereum	Go	84.33%
Erigon	Go	7.26%
OpenEthereum	Rust	5.77%
Nethermind	C#	1.78%
Besu	Java	1.22%

Tabell II
CONSENSUS LAYER (ETH2) CLIENTS

Client	Programming Language	Distribution
Prysm	Go	38.34%
Lighthouse	Rust	33.51%
Teku	Java	16.51%
Nimbus	Nim	11.54%
Lodestar	Typescript	0.01%

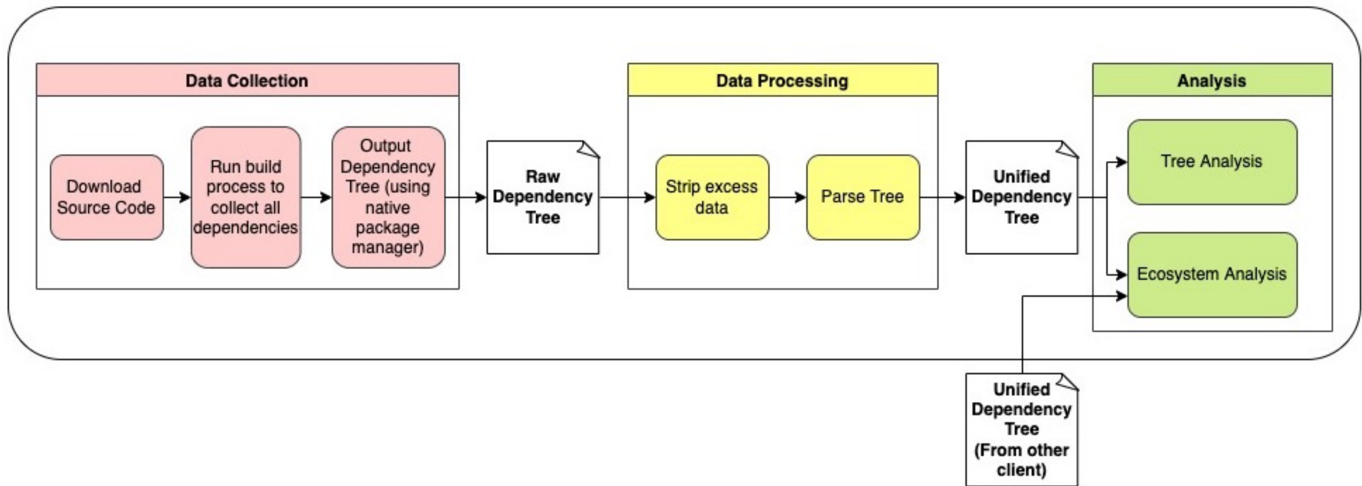
As the Ethereum blockchain is used to provide functionalities such as cryptocurrencies and decentralized finances any vulnerabilities in its software supply chain can have dire economic consequences. Mitigating this risk involves both client diversity, as a critical bug in the Eth2 consensus layer of a client used by more than 33% could cause the blockchain to go offline, as well as diversity of dependencies across the Ethereum ecosystem so that multiple clients are all affected by the same bug or malicious code injection [13].

III. RELATED WORK

In 2021 the EU Agency for Cybersecurity reported that 66% of cyber attacks target the software supply chain, and provide an outline for software consumers to navigate the growing threats [7]. Following an Executive Order in 2021 to secure the Software Supply Chains in the USA, the Cybersecurity and Infrastructure Security Agency of USA released a paper with the aim to introduce standards for software supply chains to ensure their security [7].

In 2017, Decan et al. published an analysis comparing the evolution of dependency trees in 7 popular software ecosystems. Common trends among all studied ecosystems were that they all tend to grow in number of projects and dependencies. Also, in every ecosystem the majority of projects are dependant on a minority of core projects. Differences between ecosystems included the degree of transitive dependencies, where some ecosystems remained stable while others saw an increase in the ratio of transient dependencies to direct dependencies over time. The paper also introduced new quantitative metrics to determine the *health* of ecosystems [6].

One major contribution to the study of software supply chains and software ecosystems is the Maven Dependency Graph. Presented in a 2019 paper, the Maven Dependency



Figur 1. Data pipeline for the analysis of the software supply chain of a single Ethereum client.

Graph is a snapshot of the Maven Central Repository, a package manager for Java projects. The dependency graph contains 2,4 million artifacts and 9 million dependencies, stored in a graph database with an accompanying API for querying the data set [14].

Studying software supply chains in PyPI, the package manager for Python projects, Benthal et. al introduced a model for identifying *hot spots* of risk in ecosystems [15]. These *hot spots* are determined through static analysis and show projects that is highly connected to the rest of the ecosystem through reverse dependencies, and is exposed to a large number of vulnerabilities through their dependencies.

In 2020, Ohm et al. released a paper reviewing several malicious software packages, which have been used to attack software supply chains, and outlined the main methods used to inject malicious code into the supply chain. Most commonly malicious code is introduced through *typosquating* [16], releasing a project with a name with a slight variation to that of a prominent package [4]. The second most common way is by infecting an established project, which requires the culprit to have gained access to the project either by taking over a maintainers credentials, or becoming a maintainer through social engineering.

Conducting a longitudinal analysis of Java projects from the Maven ecosystem Soto-Valero et al. focused on the usage status of dependencies in projects. They found that bloated dependencies, that is dependencies which are not actually used by the project, in most cases remain bloated [17]. Also, they showed that bloat tends to grow over time, even in cases where developers remove direct bloat. This shows that developers seldom encounter negative implications from removing currently bloated dependencies, and that all developers who maintain projects need to be diligent in pruning their supply chain for the entire ecosystem to improve.

Pashchenko et al. studied the supply chains of 200 popular Java projects, which totaled over 10000 distinct dependencies, in order to analyse the effect of software vulnerabilities. Their study showed that 20% of known vulnerabilities are never deployed and do not pose a threat to the dependant project [18]. It was also found that 81% of vulnerabilities in a projects supply chain could be fixed by updating the vulnerable dependency to a newer version. Of the studied vulnerable dependencies 1% were halted, ie abandoned by their maintainers, and posed a severe problem for downstream projects.

In 2020 Zamani et al. conducted a review of 40 security breaches of the Ethereum blockchain. They found that vulnerabilities in clients was the 2nd most prominent cause of security incidents [19].

Aumasson et al. released a security review of the Ethereum beacon chain. They discuss that although bugs in dependencies required for cryptography procedures would have more severe outcomes to the performance of the network, äll dependencies execute code at the same privilege leveländ therefore potentially harmful [20].

In 2022 Soto-Valero et al. released the first study into the software supply chain of Ethereum clients from the Java ecosystem, and looked at the evolution of the supply chains over a one year period. They found that both the number of dependencies and suppliers increased over this period. They also found that the majority of dependencies were shared amongst both clients supply chain. [21]

In a 2022 report Enck et al. summarized 3 summits they held with organisations representing both enterprise and USA policy makers discussing challenges in securing the software supply chain. [3]. A frequently recommended security measure which was discussed and disagreed upon was that of automatic version updates of dependencies. Although security experts maintain this practice eliminates exposure to vulnerabilities, many developers argued this often led to bugs and breaking changes being introduced to their projects due to unmature code. It was also mentioned that there have been incidents where software projects had been infected with malicious code, as mentioned in [4], which shows that automatically updating is not a bullet proof strategy. Another point of discussion was the practice of providing a standardized Software Bill of Materials (SBoM).

IV. Methodology

A. Project Pipeline

In order to generate a data set depicting the software supply chain of the Ethereum ecosystem, each studied client was treated according to Figure 1 individually. For each client, the latest version of the source code was downloaded from their respective GitHub repository. The build process was then invoked in order to download all direct and transient dependencies. The client software was then run to ensure that all dependencies were fetched and that the software was functional. Using the native package manager of the client their dependency tree was output. From this step in the pipeline until the analysis of the unified dependency trees, the implementation of the procedure varied depending on the package managers used. In general terms the next step was to strip the raw dependency tree of data irrelevant to the study. Examples of irrelevant data include internal (non-third-party) dependencies as well as paths pointing to the location of dependency source files on the system. Next, the raw dependency tree was parsed. Individual packages were formatted according to the artifacts schema, and the dependency relationships were formatted according to the dependency schema, both defined in IV-B. This process differed depending on the format of the raw dependency trees, which were either nested trees or lists of package pairs. Once the data was structured in the unified dependency tree format, the same procedure for analysis was utilized for all clients. Individual trees were analysed in order to collect metrics defined in section IV-C. Unified Dependency Trees of clients developed in the same programming language were also analysed together in order to collect data regarding to the intersection of their dependencies. Details of differences in implementation, and technical difficulties, of clients are described below.

1) Go: Package management in Go differs from most other programming languages in that it does not utilize a third-party package manager, nor a central repository to host software packages. Where packages are hosted is instead left up to the supplier. From manual inspection of the dependencies in the studied Go clients, GitHub is the most common hosting solution. Go does not use differentiate dependencies by scope, rather all dependencies are compiled when build procedures are invoked. In order to define dependencies in a Go project, a developer lists each dependency by the URL which points to where the package is hosted followed by its version. This is done in a file named `go.mod` in the project directory. The command `go mod graph` will output a list of all dependencies in the project, including internal non-third-party dependencies, with each line containing a dependent package and its dependency separated by a space. In order to remove internal dependencies, the command `go list -m` is used to curate a list of third party packages. These lists are used together to ensure that the unified dependency tree only consists of third party dependencies.

2) Rust - Cargo: The Ethereum clients written in Rust all use the Cargo package manager. All dependencies for a project are listed in a file named `Cargo.lock`. The dependency tree for a Cargo project can be output using the `cargo tree` command. The output of this command is a nested tree structure.

3) Java - Gradle: The Ethereum clients written in Java all use the Gradle package manager. Projects using gradle lists all dependencies in a file named `build.gradle`. The dependency tree is output using the command `gradle -q dependencies`. The output consists of several nested trees, separated by the scope of the dependencies. The output also includes non-third-party dependencies, however these were easily identified through machine-readable means and removed systematically.

B. Unified Dependency Tree

As seen in the end of section 1, the collection of data regarding a projects software supply chain is not trivial, and differs greatly between package managers. In order to facilitate easier analysis of supply chains across several software ecosystems a uniform format for this data is desirable. Also, as efforts are being made to introduce a the practice of providing software bill-of-materials (SBoM) in the software industry, a standardised model for supply chains is needed [3]. In this report, the notion of a Unified Dependency Tree is introduced. The model defines data structures using json format, as it is structured text-based format which is both human-readable and recognized by several scripting languages [22]. The model represents software packages using the data type *Artifact*. This structure has four key-value pairs which are, artifactId, which is the name of the package, groupId, the supplier of the package, version, and finally the gav, which is a concatenation of the first the values. The gav is used as the unique identifier for the artifact. All the dependencies for a project are stored in a json object. The unique id gav of each dependent artifact in the project is entered into this object and points to a list of the gav of all its dependency artifacts.

C. Metrics

For the analysis of individual clients the following metrics were collected

- Total dependencies
- Unique direct dependencies
- Unique transient dependencies
- Unique suppliers

Total dependencies is the sum of all dependencies. A package which is a dependency of several packages is counted once for each dependent package. If a dependent package is featured multiple times, its dependencies are only counted once for the dependent package.

Unique dependencies is the sum of all packages which the client is directly dependent upon as declared in their source code according to the methods described in section IV-A.

Unique transient dependencies are the sum of all packages which are featured in the dependency tree, but not directly dependent. Packages which are dependencies to several packages are only counted once.

Unique suppliers are the sum of all suppliers of packages featured in the dependency tree. Suppliers who provide more than one package, or who supply a package which is featured multiple times, are only counted once.

V. Results

RQ1. What is in the supply chain of Ethereum Clients?

After analyzing all Ethereum clients individually, there is an evident size difference between the supply chains of the two Ethereum layers as shown in Table III. Besides Besu having more unique direct dependencies than Teku, the supply chain metrics gathered are much larger for the Eth2 clients compared to the Eth1 clients of the same ecosystem. The biggest difference is seen in the Go ecosystem, in which the metrics of the Eth2 client Prysm is at least twice the size of the metrics of the Eth1 client Geth.

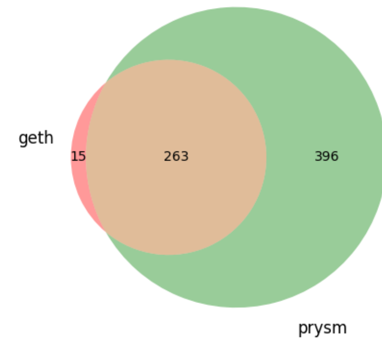
All the Eth1 clients require roughly the same amount of unique direct dependencies, while having vastly different amounts of unique transitive dependencies. For Geth (Go), there are 3.2 unique transient dependencies per unique direct dependencies; for OpenEthereum (Rust) this ratio is 5.7; for Besu (Java) it is 3.0. Although the Eth2 clients have vastly differing amounts of unique direct dependencies, the ratio of unique direct dependencies to unique transient dependencies are similar to the Eth1 clients from the same ecosystem; Prysm (Go) 4.3; Lighthouse (Rust) 5.6; Teku (Java) 3.0.

There are no clear patterns between the number of suppliers in the different Ethereum layers. There are however clear similarities between the number of suppliers per unique dependencies between clients written in the same ecosystem. For the Go ecosystem each supplier provides on average 1.7 and 2.0 unique artifacts for Geth and Prysm respectively. For the Rust ecosystem each supplier provides on average 1.5 and 1.3 unique artifacts for OpenEthereum and Lighthouse respectively. For the Java ecosystem this value is 2.7 and 2.4 for Besu and Teku respectively.

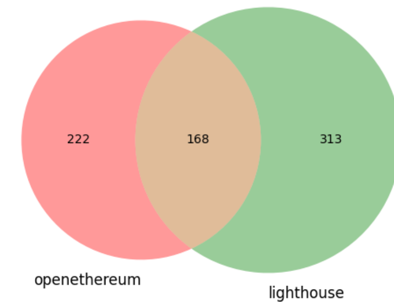
RQ2. What is the diversity of the software supply chain of Ethereum across ecosystems?

Looking at figures 2, 3, and 4, there is a drastic difference in software diversity between the studied ecosystems. For Go, shown in figure 2, there is nearly a complete overlap of unique dependencies. 95% of the unique dependencies which are required by Geth are also dependencies of the Prysm client. Of the overlapping dependencies the largest providers of monoculture are btcsuite, influxdata, mattn, and prometheus. Btcsuite provides a collection of tools for Bitcoin and cryptography in Go. Influxdata are an enterprise grade software provider which specializes in platforms for time series databases. Prometheus is also a provider of monitoring and time series database tools. Mattn is the largest provider of monoculture who is a sole developer.

The supply chains of the Java Ethereum clients are more diverse, as seen in figure 4, although the majority of unique dependencies are shared amongst both Teku and Besu. 69% of the unique dependencies in Teku, and 60% of the unique dependencies in Besu, are overlapping. The largest monoculture providers in the Java supply chain are Netty, OpenTelemetry, and Apache. Netty is by far the largest supplier for the Java clients, providing 30 dependencies which are required by both Teku and Besu. Netty provides APIs and tools for developing asynchronous server communication. OpenTelemetry is a provider of APIs for monitoring and logging data. They are the second largest supplier of dependencies, providing 12 dependencies for both clients. Apache is a vast organization and one of the most prominent contributors of open source software and support a large number of various projects. Notable packages which Apache provides both Teku and Besu are Tuweni, which aids development of cryptography functions, and log4j, a utility for creating customized log messages for running processes.



Figur 2. Intersection of Go Ethereum Dependencies



Figur 3. Intersection of Rust Ethereum Dependencies

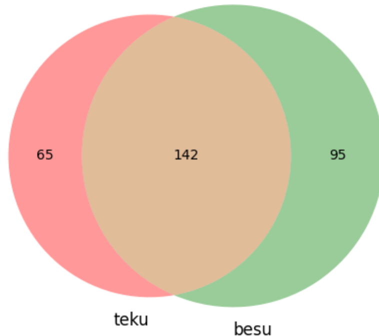
The Rust Ethereum clients have the most diverse supply chain. The majority of unique dependencies are not in the intersect of OpenEthereum and Lighthouse. Only 43% of OpenEthereum dependencies and 35% of Lighthouse dependencies are overlapping. There are very few providers of monoculture in the Rust developed clients. The most prevalent providers in Rust are Crossbeam, Parity, and Serde. Crossbeam is a provider of tools for concurrent programming in Rust. Parity are providers of numerous tools used in blockchain development in Rust. Serde is a set of tools used for serializing and deserializing data structures, which is used for storing data or transferring data over networks.

VI. Discussion

There is an apparent difference in size of the software supply chains of the Eth1 Execution layer and Eth2 Consensus layer of the Ethereum Blockchain. Although Eth1 was released 5 years before Eth2, and studies have shown that a project's dependencies tend to grow over time, the dependency metrics of the Eth2 clients are all larger than the Eth1 clients from the same ecosystem, save for Besu. This points to that the Eth2

Tabell III
SOFTWARE SUPPLY CHAIN METRICS OF ETHEREUM CLIENTS

Client	Programming Language	Total Dependencies	Unique Direct Dependencies	Unique Transitive Dependencies	Suppliers
Geth (Eth1)	Go	362	67	211	166
OpenEthereum (Eth1)	Rust	1447	67	382	299
Besu (Eth1)	Java	1473	63	149	80
Prysm (Eth2)	Go	901	123	536	328
Lighthouse (Eth2)	Rust	2044	77	435	387
Teku (Eth2)	Java	2998	59	178	99



Figur 4. Intersection of Java Ethereum Dependencies

Consensus layer is a more complex system of cryptography procedures, requiring far more external software packages in order to function. This idea is supported best by the findings shown in figure 2, which shows that 95% of the supply chain of Geth only constitues 40% of the supply chain of Prysm.

The gathered data also shows interesting trends between different ecosystems. As the clients within each Ethereum layer are functionally equal on a macro level, the study of their supply chains should reflect well on the ecosystems which they are built upon. From this study we can see that Java, the oldest of the studied ecosystems, is dominated by larger organisations who supply large amounts of Open Source software. This is assumed to be due to the effects of *hype driven development* over a long period of time. As time progresses, developers tend to choose software packages supplied by reputable vendors, and larger reputable organisations outlast and take over development from smaller vendors. Although Rust and Go are only released a year apart, 2010 and 2009 respectively, Rust is much more diverse in both software packages and suppliers. It is assumed that this is due to Go being maintained by a Google, a large corporation with more stringent software requirements, compared to Rust which is community driven.

It is self admitted that the Ethereum Foundations ambitions to achieve client diversity is far off the mark, however the data gathered in this paper points to that the software diversity is in a far worse state.

VII. CONCLUSSION

In this paper, the first systematic analysis of the software diversity, with a focus on open-source software dependencies, in the Ethereum ecosystem is presented. The dependency trees of three pairs of Ethereum clients, developed in the languages Go, Rust, and Java, were collected and transformed into a unified format. In this unified format, the dependency trees of the clients were analysed individually as well as together with clients developed in the same language. This analysis resulted in a novel data set of quantitative metrics describing this size of the Ethereum software supply chain. The data set shows that the Eth2 consensus layer requires a far greater amount of dependencies to function compared to the Eth1 execution layer. The data set is also used to show that there is a significant overlap of dependencies used by clients developed in the same language. This overlap was largest in the Go developed clients, where 95% of dependencies of the Eth1 client Geth were also dependencies of the Eth2 client Prysm. The smallest overlap seen was between the Rust clients OpenEthereum and Lighthouse, which shared 43% and 35% of their dependencies respectively.

ACKNOWLEDGEMENT

The author would like to thank the supervisors of this project, Benoit Baudry and César Soto-Valero, for their unwavering support, guidance, and words of encouragement throughout this project.

REFERENSER

[1] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, pp. 1–10, 2021.

[2] "2019 state of the software supply chain," Sonatype, Tech. Rep., 2019. [Online]. Available: https://www.sonatype.com/hubfs/SSC/2019%20SSC/SON_SSSC-Report-2019_jun16-DRAFT.pdf

[3] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security Privacy*, vol. 20, no. 2, pp. 96–100, 2022.

[4] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

[5] J. Yang, Y. Lee, and A. P. McDonald, *SolarWinds Software Supply Chain Security: Better Protection with Enforced Policies and Technologies*. Cham: Springer International Publishing, 2022, pp. 43–58. [Online]. Available: https://doi.org/10.1007/978-3-030-92317-4_4

[6] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, 02 2019.

[7] "Understanding the increase in supply chain security attacks," *ENISA*, Jul 2021. [Online]. Available: https://www.enisa.europa.eu/news/enisa-news/understanding-the-increase-in-supply-chain-security-attacks

[8] F. B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, pp. 565–584, 1993.

[9] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," 06 1997, pp. 67–72.

[10] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48, no. 1, sep 2015. [Online]. Available: https://doi.org/10.1145/2807593

[11] "The open source definition," April 2007. [Online]. Available: https://opensource.org/osd

[12] "What is ethereum?" Feb 2022. [Online]. Available: https://ethereum.org/en/what-is-ethereum/

[13] "Client diversity," Feb 2022. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/client-diversity/

[14] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 344–348. [Online]. Available: https://doi.org/10.1109/MSR.2019.00060

[15] S. Benthall, T. Pinney, J. Herz, and K. Plummer, "An ecological approach to software supply chain risk management," 01 2016, pp. 130–136.

[16] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020, pp. 509–514.

[17] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1021–1031. [Online]. Available: https://doi.org/10.1145/3468264.3468589

[18] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2018.

[19] E. Zamani, Y. He, and M. Phillips, "On the security risks of the blockchain," *Journal of Computer Information Systems*, vol. 60, no. 6, pp. 495–506, 2020. [Online]. Available: https://doi.org/10.1080/08874417.2018.1538709

[20] J.-P. Aumasson, D. Kolegov, and E. Stathopoulou, "Security review of ethereum beacon clients," *arXiv preprint arXiv:2109.11677*, 2021.

[21] C. Soto-Valero, M. Monperrus, and B. Baudry, "The multi-billion dollar software supply chain of ethereum," 2022. [Online]. Available: https://arxiv.org/abs/2202.07029

[22] "Working with json," April 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON