Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Metric-based Analysis to Identify Anomalous Releases in the npm Ecosystem Software Supply Chain

Michele Frattini

Master's Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35 16146 Genova, Italy
https://www.dibris.unige.it/

Computer Science MSc
Software Security and Engineering Curriculum

Metric-based Analysis to Identify Anomalous Releases in the npm Ecosystem Software Supply Chain

Michele Frattini

Advisors: Luca Caviglione, Giovanni Lagorio
Examiner: Matteo Dell'Amico

July, 2026

Abstract

The Open-Source Software ecosystem relies heavily on package managers such as Node Package Manager (npm), which hosts over two million libraries. However, its open nature makes it vulnerable to Software Supply Chain Attacks (SSCAs), where malicious code is injected into legitimate components. Existing detection tools share two limitations: they analyze only the most recent release, lacking a longitudinal perspective, and incur high operational costs through sandbox execution or commercial Large Language Model inference.

This thesis proposes a metric-based approach to identify anomalous npm releases by tracking their evolution across versions. We first study four SSCAs that affected the npm ecosystem in 2025: a malicious Windows library injection, a cryptocurrency-stealing attack, and two *Shai-Hulud* self-propagating worm variants. From these, we define five metrics: *File Types*, *Package Size*, *Longest Line Length*, *Unique Hexadecimal Values*, and *Unique Ethereum Addresses*. We evaluate these metrics against a goodware baseline of 629 popular packages across their 20 most recent versions, and a malware collection of 20 compromised releases.

Results show that each attack is detectable by at least one metric, but no single one identifies all four simultaneously, highlighting the value of combining lightweight measures as a practical complement to semantically rich approaches.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Open-Source Software (OSS) ecosystem is one of the core pillars of the modern software landscape, accounting for millions of active projects using publicly available libraries and components. Among these environments, the Node Package Manager (npm) registry for JavaScript has emerged as one of the largest and most dynamic ones, hosting over two million packages [Moo+21]. Here, developers can collaborate by distributing their code without having to reinvent the wheel. Integrating an external component in an application implies including not just its functionality, but also trusting the original authors, all transitive dependencies, and the infrastructure involved in the distribution process. These elements collectively form the Software Supply Chain (SSC) [GAH23].

Compromising any part of this chain affects all projects that depend on it, potentially reaching millions of users and applications. An attack targeting the SSC falls into the category of a Software Supply Chain Attack (SSCA), representing a rising threat to the security of the software ecosystem. Indeed, in the second half of 2025, four major incidents took place within npm, including five packages compromised in July, like `eslint-config-prettier` with over thirty million weekly downloads [Lab25]; eighteen others hijacked in September to steal cryptocurrency [HH25], like `chalk` and `debug` collectively installed over 2.6 billion times per week; the `Shai-Hulud` worm that infected 187 modules in the same month to extract the credentials of developers [BC25], and lastly a more aggressive version, infecting over a thousand components within hours in November [Cut25].

Several methodologies and detection systems have been proposed to counter these threats, spanning from static analysis (e.g., `CodeQL` [Zah+25]) through dynamic execution frameworks (like `DONAPI` [Hua+24a]), to Large Language Model (LLM)-based solutions (such as `SpiderScan` [Hua+24b] and `SocketAI` [Zah+25]). However, these tools share limitations, most notably the lack of longitudinal perspective, as they analyze only the most recent version of a package in isolation, ignoring the evolutionary history of its releases.

This, combined with high operational costs, makes continuous registry-scale monitoring challenging.

The present thesis work adopts a complementary position in this overview, evaluating five lightweight, easily computable metrics extracted directly from tarball artifacts. By tracking their evolution across 20 consecutive versions, their effectiveness is demonstrated against a baseline of 629 popular goodware samples and a malware dataset of 20 packages compromised by the four SSCAs. The results indicate that while each individual attack is detectable by at least one metric, no single one identifies all four simultaneously. This outcome reflects the structural diversity of the malicious payloads, which leave distinct traces that no single measurement can fully capture. The primary contribution of this work is to demonstrate that a metric-based, multi-version monitoring strategy provides a practical and low-cost addition to existing semantically rich methodologies.

The remainder of this thesis is organized as follows. Chapter 2 provides the foundational concepts underlying this work, introducing the SSC and the main SSCA vectors, the architecture of the npm ecosystem, and the structure of tarballs. Chapter 3 surveys existing detection approaches, spanning static analysis, dynamic analysis, LLM-based methods, and metric-based analysis, and positions this thesis with respect to their limitations. Chapter 4 presents and analyzes in depth the four SSCAs that affected the npm ecosystem in 2025, which serve as case studies throughout the research. Chapter 5 describes the two datasets constructed for the evaluation, the goodware and the malware dataset, together with the two software tools developed to build them. Chapter 6 defines the five metrics, analyzes their distribution in the goodware baseline, and evaluates their effectiveness in detecting the compromised releases. Finally, Chapter 7 summarizes the findings, discusses the limitations of the proposed approach, and outlines directions for future work.

Chapter 2

Background

This chapter provides the foundational concepts needed to understand how the SSC works and how attackers exploit it. Section 2.1 introduces the notion of SSC, describes the principal attack vectors through which SSCAs are carried out, examines the categories of victims typically targeted, and discusses source code obfuscation techniques used by both owners, to protect intellectual property, and attackers, to hide malicious code. Section 2.2 presents the npm ecosystem, the package registry at the center of all attacks considered in this work, and Section 2.3 describes the internal structure of npm tarballs, the artifacts distributed to end users and the primary subject of analysis.

2.1 Software Supply Chain and Attacks

Modern software development rarely starts from scratch. Developers routinely incorporate ready-made components into their projects to handle common functionalities such as date parsing, HTTP request handling, or UI rendering, rather than reimplementing them. These reusable units, commonly called *packages*, can in turn depend on other components, forming part of a broader ecosystem of software artifacts, tools, processes, and human actors involved in their lifecycle, collectively referred to as the SSC [GAH23].

Such components are distributed through open-source registries, such as npm, Python Package Index (PyPI) and RubyGems, which together host millions of software modules used by developers worldwide. The decentralized and open nature of these repositories is one of their greatest strengths, as anyone can freely publish a package; however, this openness also constitutes a significant attack surface [Ohm+20]. An attacker who manages to inject malicious code into a legitimate package update can compromise all the projects that transitively depend on it. This category of threat to the SSC is known as a SSCA,

and has become increasingly prevalent in recent years [Ohm+20]. Notably, over 245,000 malicious packages were discovered in the npm and PyPI ecosystems in 2023 alone, a figure that doubles the total of all those found in the previous four years combined [Hua+24b]. This trend is not merely statistical. As the attacks analyzed in Chapter 4 will illustrate, a single compromised release of a widely used package can affect millions of end users within hours. Attackers typically employ techniques such as *Account Compromise*, *Typosquatting* and other methods detailed in Section 2.1.1 to trick developers into integrating harmful code.

2.1.1 Attack Vectors

SSCAs can be conducted through different attack vectors, each exploiting distinct vulnerabilities in the ecosystem. Here, we describe the main vectors identified in the literature [Dua+20], which primarily target upstream stakeholders, such as *Package Maintainers (PMs)*, the developers responsible for publishing and managing a given package, and *Registry Maintainers*, the registry administrators. Compromising either one gives an attacker a privileged position to push malicious code to all downstream users.

The most common vector is *Account Compromise* (also known as *Account Hijacking*), which consists of gaining unauthorized access to the registry account of a PM. Once the attacker controls the account, they can publish new versions of any package managed by that maintainer, embedding malicious code that will be distributed to all downstream users the next time they run an update or start from a fresh installation. This vector is particularly insidious because the malicious version reaches users through the same trusted channel they normally rely on, as is the case for all four SSCAs analyzed in Chapter 4.

A similarly prevalent attack vector is *Typosquatting*, in which an attacker registers a package whose name closely resembles that of a popular, legitimate one, differing by only one or two characters, in the hope that developers will accidentally install it instead of the intended target. For instance, the malicious package `crossenv` was published to mimic `cross-env`, exploiting the common omission of the hyphen [Sny21]. A variant of this technique, known as *Package Masking*, involves registering a module with a name identical to a popular one from another registry, relying on the developer assumption of cross-ecosystem availability when in fact the component is not present in the target platform. For instance, in 2025 a malicious package named after the npm package `colorizr` was uploaded to PyPI [Mey25].

Another vector is *Direct Publication*, in which an attacker simply publishes a new malicious package without relying on typos or account hijacking. This can serve purposes such as *bot tracking*, where the package sends data to the server of the attacker upon installation to map the number and types of environments where it is installed, or *malware hosting*, where the tarball acts as a vehicle for additional malicious payloads that are downloaded

and executed on the victim machine [Bal19]. Both are made possible by the openness of these registries, where anyone can freely publish without prior review.

Ownership Transfer exploits the open-source practice of transferring package ownership to new PMs. Abandoned projects can be reclaimed by new parties, or legitimate owners can be persuaded through social engineering into handing over control, after which the successor can publish compromised versions. A notable instance is the `event-stream` incident of 2018, in which a new maintainer added a malicious payload to the abandoned package targeting a Bitcoin wallet, which went undetected for over a month [Tec18].

Malicious Contributor and *Disgruntled Insider* represent vectors that originate from within the package development process itself. In the first case, a seemingly legitimate contributor submits a pull request that introduces vulnerable or malicious code without the knowledge of the PMs. A prominent example is the `XZ Utils` backdoor, in which a contributor known as Jia Tan spent nearly two years building trust before inserting a backdoor into `liblzma` package [Aka24]. In the second, an authorized maintainer deliberately sabotages the project they are responsible for. For instance, in early 2022 the maintainer of the `colors` and `faker` packages intentionally introduced breaking changes as a form of protest, impacting thousands of dependent applications [Ble22].

Finally, *Registry Exploitation* refers to directly exploiting a vulnerability in the platform itself (e.g., through a remote code execution flaw), allowing attackers to modify or replace packages directly. Unlike the other vectors, this attack does not rely on compromising any individual stakeholder, but rather targets the central infrastructure shared by the entire ecosystem. It is, however, considerably harder to execute and comparatively rare in practice.

2.1.2 Victims and Impact

Not all parties in the ecosystem are equally exposed to SSCAs. In this section, we examine the two main victim categories identified in the literature [Dua+20] and discuss the potential impact on each.

The first category consists of *developers and maintainers*, who may be targeted directly to steal credentials, tokens, or cloud service access keys. Additionally, their infrastructure and machines may be compromised as part of the attack. The second category consists of *end users* of applications that depend on malicious packages, who may have their sensitive data stolen or their devices compromised. In the most severe cases, a single package with a large number of weekly downloads can expose hundreds of millions of users to risk within hours of the malicious version being published, as illustrated concretely by the attacks analyzed in Chapter 4.

2.1.3 Code Obfuscation

Since packages are often distributed as source code or build artifacts, as described in detail in Section 2.3, a PM has full control over the content before publication. This flexibility, while legitimate when used to protect intellectual property, also enables malicious actors to inject harmful code and conceal its behavior from automated scanners and human reviewers. The practice of deliberately transforming code to make it harder to understand while preserving its functional behavior is known as *code obfuscation*. Although obfuscation techniques can theoretically be applied to any programming language, this work focuses specifically on JavaScript because the npm ecosystem is fundamentally built around JavaScript and Node.js, making `.js` files the most abused and critical attack surface within the SSC. Various obfuscation techniques have been documented in the literature and observed in real-world attacks [Ohm+20; Moo+21].

The most widespread technique consists of encoding identifiers such as variable and function names using hexadecimal notation, replacing meaningful names with opaque tokens [Ohm+20]. For example, a variable name may appear as `_0x26499F`, where the underscore is prepended because JavaScript identifiers cannot start with a digit, while numeric constants appear directly as `0x1c8` [Cona]. This convention is observed in the SSCAs analyzed in Sections 4.3 and 4.5. The result is source code that is syntactically valid but semantically opaque, making manual inspection very difficult and effectively hiding malicious behavior from static analysis.

Other common techniques include *string sampling* [Ohm+20], in which seemingly random strings are reconstructed dynamically to hide sensitive data (e.g., URLs, API keys); *dynamic code generation* [Moo+21], which leverages the dynamic nature of JavaScript to execute code generated at runtime (e.g., with `eval()`); and *dead-code injection* [Moo+21], which floods source files with irrelevant instructions to increase noise and hinder manual review.

2.2 The npm Ecosystem

The npm registry is the de facto standard platform for distributing JavaScript and Node.js software, hosting over two million entries [Moo+21], and understanding its architecture and security properties is essential for analyzing how attacks propagate through the ecosystem. This section draws from the official documentation [npm26], while Figure 2.1 provides a high-level overview of the workflow involved in publishing and consuming artifacts.

The public registry is a database where developers can share and download packages, i.e., self-contained folders of files that can be incorporated into other projects. To resolve a package by name and version, npm communicates with the registry endpoint (configured by

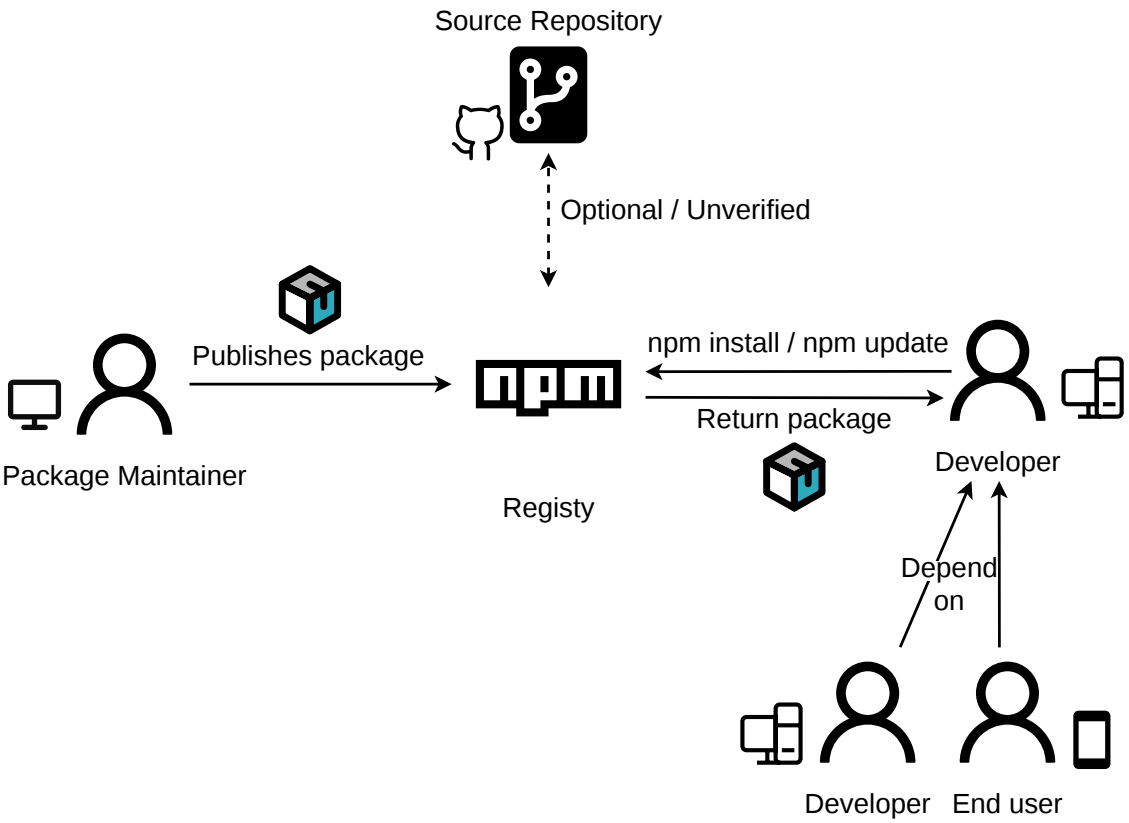

Figure 2.1: High-Level Overview of the npm Distribution Workflow.

default at https://registry.npmjs.org), from which any available release can be retrieved. When publishing, the maintainer uploads a *tarball*: a gzipped archive representing the code at a specific version and constituting the actual artifact distributed to users. Once published, it becomes available for download and installation by other developers, who integrate it into their projects to be used by end users and further downstream stakeholders, as previously described in Section 2.1 and illustrated in Figure 2.1.

Although a package may reference a Git URL pointing to a source repository, a connection represented by the dashed arrow in Figure 2.1, two critical aspects must be considered. First, the content of the tarball may differ from that of the repository, even if both nominally represent the same version. This discrepancy occurs because maintainers can selectively include files in the tarball through two mechanisms: the optional `files` field in `package.json` explicitly lists which files to include, and the `.npmignore` file (analogous to `.gitignore`) specifies which files to exclude. As a consequence, to ensure a consistent

analysis across packages, it is necessary to examine the same type of artifact rather than mixing tarballs with repository contents. Second, a crucial security gap [Dua+20] is that providing a link to a public source repository is optional in `package.json`. When no such link is present, there is no source of truth against which the tarball contents can be verified. Even when a link is provided, the registry performs no such verification, meaning that an attacker can inject malicious code into the tarball at publication time without modifying the corresponding repository.

An equally important property of the npm registry is that once a package is published under a given name and version, that combination can never be reused, even if the release is subsequently removed with the `npm unpublish` command. This means that a malicious update, once published, cannot be overwritten by the legitimate maintainer. The typical response is to unpublish it and issue a new, corrected one, often with the same content as the last legitimate version that preceded the attack; alternatively, developers can downgrade to that same prior stable release.

2.3 Tarball Content

While a npm tarball can theoretically contain any type of file, in practice, most of them follow a consistent structure, as illustrated in Figure 2.2. Understanding the typical contents of a tarball is essential to identifying potential injection points for malicious code. This section describes the main file categories typically found in npm packages [Pin+23].

Required for publication to the npm registry, every valid package must include a `package.json` file that describes its contents and properties. Its `name` and `version` fields are mandatory and together form a unique identifier. Other common optional fields include a description, the license, a link to the source repository, and the list of dependencies on other packages. As noted in Section 2.2, since no repository link is required, the registry does not verify the correspondence between the tarball and its declared source code, creating a significant security gap [Dua+20].

Most packages include documentation files such as `README.md`, `CHANGELOG.md`, and a `LICENSE` text file specifying copyright terms. These files carry no executable logic and are included purely for informational purposes, providing human-readable descriptions of the package functionality, usage instructions, and licensing conditions.

The core of a package consists of JavaScript and TypeScript files defining its functional logic. In addition, tarballs can include *build artifacts*, that is, files that are automatically generated by build tools rather than written directly by developers [Gos+20]. Through processes such as *minification*, which strips superfluous elements (such as whitespace, comments, and long variable identifiers) without altering behavior [MDN], and *bundling*, which

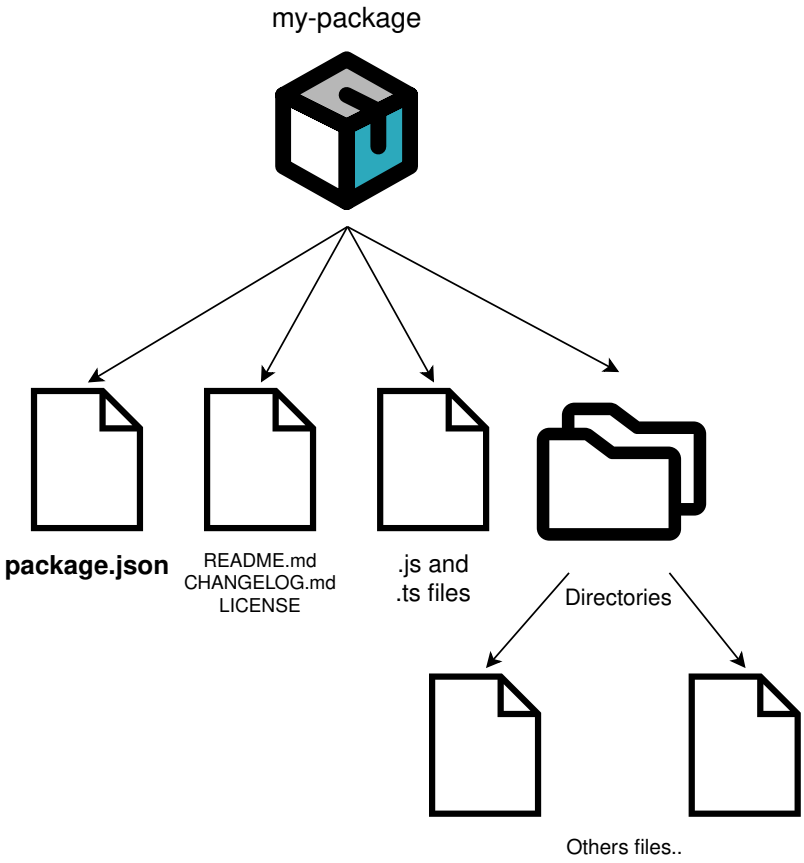

Figure 2.2: Example of a Typical npm Tarball Directory Structure.

merges multiple files into a single self-contained one, to simplify the distribution [Conb], these tools produce optimized versions of the original source code that are smaller and faster to load. A common example is source map files (identifiable by the `.map` extension), which map the minified output back to the original source to facilitate debugging [Yee]. Build processes also ensure compatibility, as the resulting code can run across different environments and runtimes without the end user needing to worry about the original syntax.

A significant practical implication of this flexibility is that the npm ecosystem provides no standardized way to distinguish source files from build artifacts, nor to locate them within the tarball. While common conventions suggest that directories such as `dist/`, `build/`, or `lib/` contain generated files, this structure is neither guaranteed nor verifiable in advance, as it depends entirely on the choices of the individual developer.

Chapter 3

Related Work

The detection of malicious packages in open-source registries has attracted considerable research attention in recent years, and the proposed approaches can be broadly grouped into four categories: static analysis, dynamic analysis, machine learning and LLM-based methods, and metric-based analysis. Section 3.1 examines representative static analysis tools, while Section 3.2 covers dynamic analysis approaches. Section 3.3 describes in detail the two most representative LLM-based tools, `SpiderScan` and `SocketAI`, and Section 3.4 surveys metric-based methods that rely on measurable structural properties of packages. Finally, Section 3.5 discusses the two main limitations affecting most existing tools which motivate this research, and clarifies how this thesis relates to and differs from existing work, highlighting the contribution it makes to the field.

3.1 Static Analysis

Static analysis examines the source code and metadata of a package without executing it. Its main advantage is speed, as packages can be screened rapidly and at scale, because no sandboxed execution environment is required. The core idea behind most static analyzers is *source-to-sink data-flow analysis* [Zah+25]: the tool identifies *sources*, points in the code where external or sensitive data enters the program, and *sinks*, points where that data could be used in a dangerous way, such as being sent over the network, written to disk, or passed to an execution primitive. A package is flagged as suspicious when data flows between sources and sinks without proper sanitization. This abstraction is powerful because the analyzer reasons over a static representation of possible execution paths, typically relying on a predefined set of sensitive Application Programming Interfaces (APIs) or an abstract syntax tree, to determine whether dangerous flows can occur.

One representative tool of this family is `CodeQL`, a query-based static analysis engine, which works by extracting the static features of the code into a searchable representation, treating the codebase as a database that can be interrogated using specific queries [Zah+25]. Thirty-nine custom `CodeQL` rules were used to flag malicious JavaScript patterns in npm packages [Zah+25], covering behaviors such as credential exfiltration through network APIs, connections to suspicious domains, unauthorized file system access, and the execution of dynamically generated code via sinks like `eval()`. `CodeQL` serves as a pre-screening step within the `SocketAI` workflow described in Section 3.3.2, reducing the number of files that need to be forwarded to the more expensive LLM-based analysis stage.

A fundamental limitation of purely static approaches is their difficulty with obfuscated code, as discussed in Section 2.1.3. When dangerous behavior is hidden behind some forms of encoding, it cannot be matched against a fixed set of syntactic rules, causing static analyzers to miss the threat entirely. A second limitation is that static analysis works on an approximation of the program: since the code is never actually executed, the analyzer may flag flows that would never trigger at runtime, producing false positives, or overlook behaviors that only emerge from values computed dynamically, producing false negatives. These trade-offs motivate hybrid approaches described in the following sections.

3.2 Dynamic Analysis

Dynamic analysis overcomes the obfuscation limitation by executing the package in a controlled environment and observing its runtime behavior. The primary cost is the significant overhead of setting up and tearing down sandbox environments for every candidate package, as well as the risk that some malicious payloads may not be triggered during the bounded execution window.

`DONAPI` [Hua+24a] is a representative tool that combines static and dynamic analysis in a two-phase pipeline. In the static phase, the entry-point files of a package and their dependencies are analyzed to extract behavioral features based on API call sequences. Rather than checking for isolated dangerous functions, `DONAPI` focuses on the order in which APIs are invoked, capturing combinations characteristic of adversarial activity. The authors identified twelve specific indicators describing typical illicit behaviors, such as sending sensitive information to an external server or downloading and executing code dynamically. These features are used to assign a suspiciousness score and to filter which packages proceed to the more expensive dynamic phase. Candidates flagged as suspicious are then executed inside isolated Docker containers, where a custom instrumentation layer intercepts and records runtime API calls. Rather than relying on system call tracing tools such as `Strace` or `Sysdig`, which operate at the OS level and lack interpretability, `DONAPI` injects monitoring code directly into 132 Node.js API implementations, capturing both the

calls and their explicit parameters without interference from unrelated processes running on the same host.

After deployment in a real-world monitoring environment from January to June 2023, `DONAPI` identified and manually confirmed 325 previously unknown malicious packages and discovered 246 API call sequences not seen in prior samples. Despite these results, the authors of `SpiderScan` [Hua+24b] note that the heavyweight nature of the dynamic execution phase, requiring a full container per package, significantly impedes throughput and practical scalability.

3.3 Machine Learning and LLM-based Methods

The third category encompasses techniques that use machine learning models, including LLMs, to detect malicious packages and identify harmful code fragments within them. The two most representative tools in this space are `SpiderScan` and `SocketAI`, described in the following sections.

3.3.1 SpiderScan

`SpiderScan` [Hua+24b] proposes a graph-based approach that avoids the cost of full dynamic execution while capturing richer semantic information than plain static analysis. The tool operates in two phases: an offline phase to construct malicious behavior models from known samples, and an online phase to screen newly published software against those patterns.

In the offline phase, `SpiderScan` processes a set of known compromised artifacts and constructs a *malicious behavior graph (MBG)* for each distinct attack vector. Each graph node represents a sensitive API call, and edges encode two types of relationships: *control-flow dependencies*, capturing the conditions under which a call is made, and *data-flow dependencies*, which capture the variables that carry data between calls. Rather than limiting the sensitive APIs to a hand-curated fixed list, `SpiderScan` uses an LLM to recognize sensitive calls in both Node.js built-in modules and third-party dependencies, which are too numerous and rapidly evolving to be enumerated manually.

In the online phase, given a target package, `SpiderScan` constructs its *suspicious behavior graphs (SBGs)* and matches them against the *MBGs*. Multiple *SBGs* are generated because a package may contain several distinct behavioral patterns, each requiring separate graph representation. Only when a match is found does `SpiderScan` invoke dynamic analysis and an additional LLM query to confirm maliciousness, keeping the expensive components on the critical path only when strictly necessary.

In its evaluation, `SpiderScan` monitored 298,504 newly published packages in npm over three months, identifying 249 previously unknown malicious releases. A limitation noted by its authors is that LLM queries for API recognition and maliciousness confirmation carry non-trivial costs per query, which become significant at registry-level monitoring scale.

3.3.2 SocketAI

`SocketAI` [Zah+25] is a malware detection workflow leveraging `GPT-3` and `GPT-4`. The design relies on a two-part prompt structure: a *system-role* prompt sets the behavioral context for the model, instructing it to act as a security expert reviewing JavaScript code for malicious patterns, while a *user-role* prompt provides the file content to be analyzed. To contain API costs, the workflow uses `CodeQL`-based static analysis as a pre-screening step that reduces the number of files forwarded to the LLM.

Evaluated on 5,115 npm packages, of which 2,180 contain harmful code, `GPT-4` outperforms `CodeQL` alone, while the combined static-LLM pipeline achieves the best cost-accuracy trade-off overall. As with `SpiderScan`, the use of commercial LLM APIs makes continuous monitoring of the entire registry expensive, representing a shared limitation of both approaches.

3.4 Metric-based Analysis

A complementary line of research focuses on measurable, structural characteristics of source files and package artifacts, rather than on the semantics of API calls. This approach does not require execution or language model inference, making it lightweight and straightforward to apply at scale.

The relationship between software metrics and vulnerabilities was studied in [Wei+25], proposing a taxonomy organized into several families: *code smell indicators* such as the number of `goto` statements and function pointers, *complexity metrics* such as the number of loops and local variables, *memory management operation counts*, and *Abstract Syntax Tree (AST) metrics*. While their focus is on vulnerability identification rather than SSCA detection, their metric taxonomy provides a broad reference point for what structural properties can be extracted from source code.

A narrower but directly supply-chain-relevant question was addressed in [Zor25]: can simple metrics derived from the whitespace and character distribution of files detect *Blank Space Trojan (BST)* and *Hidden Unicode Trojan (HUT)* attacks? BST attacks hide malicious logic after long sequences of blank characters, exploiting parser behaviors, while HUT attacks use invisible Unicode control characters, such as right-to-left override markers, to

make the implementation appear different to a human reviewer than what is actually parsed and executed. The ratio of whitespace to printable characters was proposed as a key metric to detect such anomalies in GitHub repositories. A significant aspect of his methodology is that the analysis is performed across multiple consecutive versions of a project, studying how the metric evolves over time rather than examining a single snapshot. This multi-version perspective is shared with the present thesis, which similarly analyzes the evolution of metrics across 20 consecutive package versions to identify anomalous releases.

3.5 Positioning of This Research

Despite the breadth of existing approaches, two recurring limitations affect most of the tools discussed above.

The first is the *absence of a longitudinal perspective*. Most tools, including `DONAPI` [Hua+24a] and `SpiderScan` [Hua+24b], analyze only the most recently published version of a package, without tracking how its properties change across releases. This design choice makes it impossible to detect an abrupt anomalous change in a metric that is otherwise stable, which is precisely the signature left by the SSCAs analyzed in Chapter 4.

The second is *operational cost*. Dynamic analysis, as in `DONAPI`, requires container instantiation per candidate package version, while LLM-based methods, as in `SpiderScan` and `SocketAI` [Zah+25], incur per-query inference costs that grow with monitoring volume and introduce dependencies on external services. Together, these constraints make continuous, registry-scale monitoring challenging.

The present thesis occupies a complementary position in this landscape. Rather than conducting semantic analysis of API calls or relying on dynamic execution or commercial LLM services, it evaluates a set of lightweight, easily computable metrics derived directly from tarball artifacts, such as: *file types*, *package size*, *longest line length*, *unique hexadecimal values* and *unique Ethereum addresses*. Their effectiveness is evaluated against a baseline established from 629 popular goodware packages, and their evolution is tracked across 20 consecutive versions per package. As shown in Chapter 4, the four recent SSCAs analyzed in this work leave measurable traces in at least one of these metrics, suggesting that a metric-based, multi-version monitoring strategy can serve as a practical and low-cost complement to the more semantically rich approaches surveyed above.

Chapter 4

Analysis of Recent Attacks

This chapter presents and analyzes, in chronological order, the four SSCAs that occurred in the npm ecosystem in 2025, selected as case studies for the metric-based evaluation conducted in Chapter 6. Before describing each attack individually, Section 4.1 introduces the attack model that characterizes all four cases: the relevant threat actors, the phases of a typical attack, and the type of artifacts that are modified. Section 4.2 through Section 4.5 then analyze each attack in detail, describing the number of packages infected, the inner workings, the characteristics, structure and behavior of the malicious code, and the key differences between the compromised versions and their benign counterparts.

4.1 Attack Model

All four attacks studied in this work follow the same high-level model, illustrated in Figure 4.1. The threat actor is an external attacker who targets the npm ecosystem by exploiting the *Account Compromise*: first, the attacker obtains the credentials of a legitimate PM, typically through a phishing campaign (Step 1 in Figure 4.1), and then uses the compromised account to publish a new malicious update of the packages managed by that maintainer (Step 2 in Figure 4.1), which is automatically distributed to all downstream victims upon the next `npm install` or `npm update`. These include developers who directly depend on the package, those whose projects transitively rely on it, and end users of the software built on top of it.

The malicious version differs from its legitimate predecessor in the content of the tarball: the attacker adds one or more new files, or modifies existing ones, to embed a payload in the tarball, which is automatically activated during installation or update. The specific payloads differ across the four attacks, as illustrated in Figure 4.2, spanning remote code

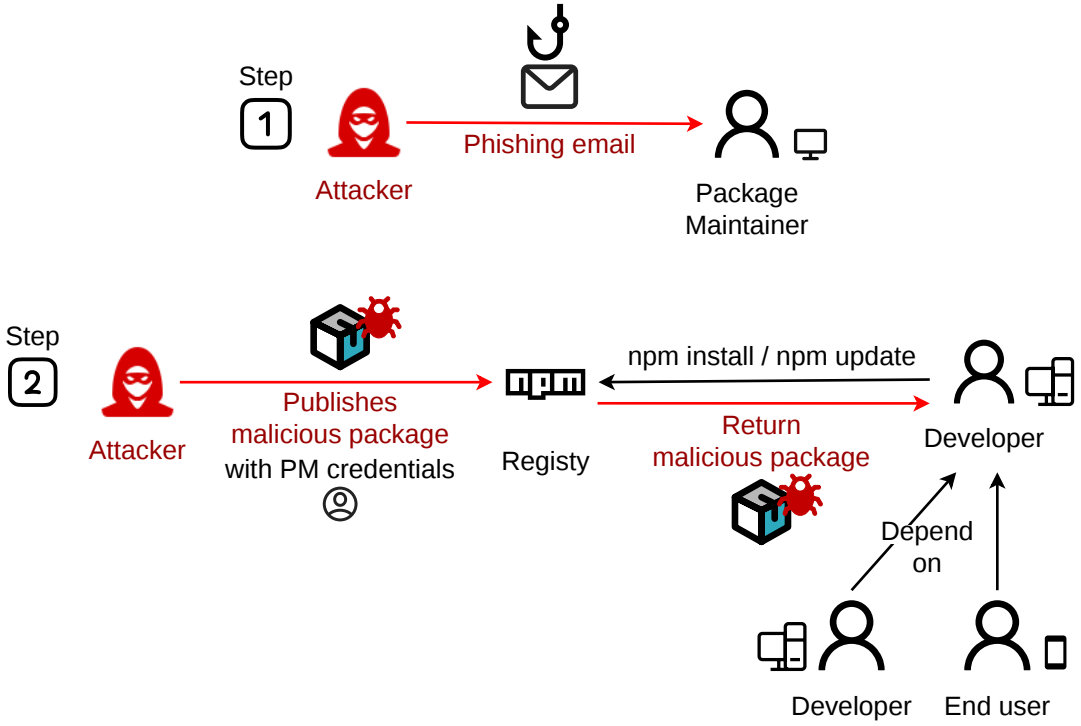

Figure 4.1: Attack Model for the Four SSCAs.

execution via a Dynamic Link Library (Section 4.2), cryptocurrency theft through browser transaction hijacking (Section 4.3), and large-scale credential exfiltration with worm-like self-propagation (Sections 4.4 and 4.5).

4.2 Attack A: Malicious Windows DLL

The first attack occurred on July 18, 2025, and compromised five npm packages. Among these, `eslint-config-prettier` is the most widely used, with over thirty million weekly downloads [Lab25]; the remaining infected packages were `eslint-plugin-prettier`, `synckit`, `@pkgr/core`, and `napi-postinstall`.

The tarball of each malicious version contains two new files, namely a script called `install.js` and a Dynamic Link Library (DLL) called `node-gyp.dll`, weighing 1.2 MB. During the installation or update of one of these compromised packages, the `install.js` script is automatically executed; it checks the operating system of the victim and, upon detecting a Windows environment, invokes `rundll32.exe` to load `node-gyp.dll` into memory, thereby enabling remote code execution and the subsequent installation of additional malware.

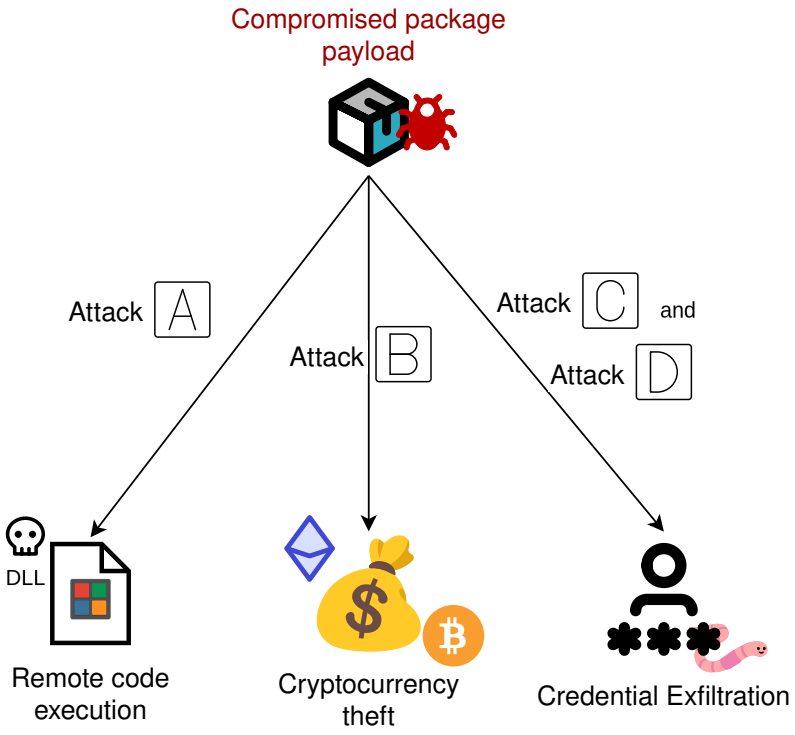

Figure 4.2: Compromised Package Payload.

4.3 Attack B: Crypto attack

On September 8, 2025, an attacker compromised eighteen popular packages, including `chalk` and `debug`, respectively the first and third most popular ones in terms of the number of stars on npm [Tri26]. Collectively, these modules account for over 2.6 billion weekly downloads, making this one of the most impactful attacks in the recent history of the npm ecosystem [Mat25]. Fortunately, the financial impact was limited, as the infected versions remained online for approximately two hours before being detected by developers and removed from the registry. The attack was made possible by compromising the account of a PM (qix); after obtaining their credentials, the attacker published malicious updates of these packages, which allowed theft of cryptocurrency by intercepting and manipulating user transactions.

The operation of the malware can be broken into three phases [HH25]:

1. **Injection into the Browser:** The malware integrates itself directly into the browser environment, hooking critical functions such as `fetch` and `XMLHttpRequest` (the core APIs responsible for web data transfer), and wallet-specific APIs such as `window.ethereum`. This allows it to intercept web traffic and wallet interactions in real time.

2. **Monitoring for Sensitive Data:** Once active, the code scans network responses and transaction payloads for cryptocurrency wallet identifiers and related metadata. It can recognize multiple formats, including Ethereum, Bitcoin, Solana, Tron, Litecoin, and Bitcoin Cash.

3. **Transaction Manipulation:** When a transaction is identified, for instance, a user is about to send cryptocurrency to a specific recipient, the malware substitutes the legitimate destination address with one controlled by the attacker. It uses visually similar replacements to make the substitution harder to detect, thereby increasing the probability of a successful theft.

   In addition to replacing the destination address, the malware can also modify other transaction parameters, such as approvals or allowances, prior to the user signing. For example, when the user interacts with a smart contract, such as to swap tokens, the malware can silently route funds to the attacker, even if the user interface appears correct.

   To achieve this, the malicious code embeds the Ethereum addresses of seven popular smart contracts:

   - *Uniswap V2*: `0x7a250d5630b4cf539739df2c5dacb4c659f2488d`
   - *Uniswap V3*: `0xe592427a0aece92de3edee1f18e0157c05861564`
   - *Uniswap V4*: `0x66a9893cC07D91D95644AEDD05D03f95e1dBA8Af`
   - *PancakeSwap V2*: `0x10ed43c718714eb63d5aa57b78b54704e256024e`
   - *PancakeSwap V3*: `0x13f4ea83d0bd40e75c8222255bc855a974568dd4`
   - *1inch*: `0x1111111254eeb25477b68fb85ed929f73a960582`
   - *SushiSwap*: `0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f`

The malicious code is contained in the `index.js` file, written as a single obfuscated line of 76,438 characters, which also increases the file size. The obfuscation technique employed encodes function names, variables, and constants in hexadecimal notation, filling the `index.js` file with opaque tokens such as `0x1c8` or `_0x6fd57a`, which hide the injected malicious logic among the meaningful identifiers present in legitimate versions. Figure 4.3 illustrates this concretely: the `index.js` file of the compromised `chalk` package shows the legitimate imports on lines 1 to 6, while line 11 contains the entire injected payload as a single long string, as visible from the tiny horizontal scrollbar at the bottom of the editor. As a result, many behavioral characteristics of the malware are hidden from static analysis, including the hooking of critical functions and wallet-specific APIs, the substitution of the transaction destination, and the wallets controlled by the attacker. The only features detectable in plain text are the seven smart contract Ethereum addresses, which are not obfuscated as they are publicly known and legitimate.

Figure 4.3: The `index.js` File of the Compromised `chalk` Package in Attack B.

4.4 Attack C: Shai-Hulud 1.0

On September 15, 2025, 187 npm packages, including `@ctrl/tinycolor`, with over two million weekly downloads, were compromised by the SSCA known as `Shai-Hulud` 1.0 [BC25]. The initial infection vector and the identity of patient zero remain unknown; phishing is considered the most likely hypothesis [Tea25a]. The goal of this attack was to steal sensitive information from developers and npm maintainers, specifically GitHub and npm authentication tokens, cloud services access keys (e.g., Amazon Web Services (AWS), Google Cloud Platform, and Azure), and local system data.

The operation of the malware can be divided into two phases:

1. **Secret Scanning:** When the victim finishes installing or updating a compromised package, a malicious script is executed and it downloads, installs, and runs a platform-appropriate version of `TruffleHog` [Sec]. `TruffleHog` is a legitimate tool normally used to prevent accidental leakage of secrets in source code repositories; in this context, however, it is used to maliciously search for secrets in the local file system of the victim.

2. **Validation and Propagation:** After the search with `TruffleHog` completes, the malware checks whether the discovered npm and GitHub tokens are valid and active. If valid npm credentials with publish privileges are found, it publishes a new malicious version for every package managed by the victim maintainer, embedding the same malicious code in each one; this self-propagation mechanism, analogous to that of a worm, explains the large number of packages compromised. If valid GitHub tokens are found, the malware creates a public repository on the account of the compromised maintainer named `Shai-Hulud` (like the sandworm in the *Dune* novels), containing a `data.json` file with all the stolen information encoded.

The malicious code is injected into the tarballs as a new file named `bundle.js`, which weighs approximately 3.7 MB and contains a single line of code 3,734,355 characters long, as illustrated in Figure 4.4 for the compromised `ember-browser-services` package.

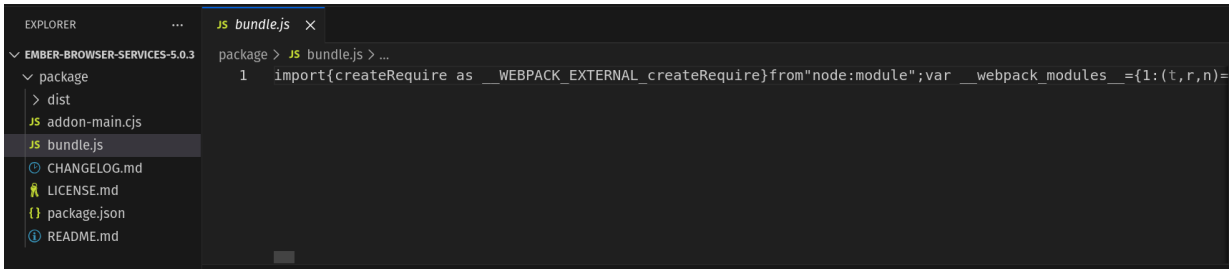

Figure 4.4: The `bundle.js` File of the Compromised `ember-browser-services` Package in Attack C.

4.5 Attack D: Shai-Hulud 2.0

Only two months after `Shai-Hulud` 1.0 (Section 4.4), on November 24, 2025, a new and more aggressive version was identified under the name `Shai-Hulud` 2.0 [Cut25]. This time, the SSCA compromised 1,055 npm packages within a few hours [Cut25]. The goal remained unchanged compared to the previous version, aiming at the theft of sensitive information such as GitHub and npm authentication tokens, access keys for cloud services, and system data. The operating mechanism was similarly preserved: as described in Section 4.4, the attack relies on the automated execution of `TruffleHog` [Sec] to search for secrets in the local file system of the victim, followed by data exfiltration through a public repository and self-propagation if valid npm or GitHub tokens are found.

However, this version introduces several technical differences and improvements over its predecessor [Tea25b]:

1. **Bun Runtime:** Rather than operating in a Node.js environment, this version executes the malicious code using the Bun JavaScript runtime [Sum26], enabling faster execution.

2. **Malicious Files:** The tarball of each compromised package contains two new files: a Bun configuration script `setup_bun.js` and the payload `bun_environment.js`, which weighs over 10 MB and contains the malicious code as a single obfuscated line of 10,157,373 characters, as illustrated in Figure 4.5 for the compromised `@asyncapi/avro-schema-parser` package. The configuration script simply calls the payload file.

3. **Obfuscated Code:** The malicious code employs an obfuscation technique that encodes function names, variables, and constants in hexadecimal notation, filling `bun_environment.js` with opaque hexadecimal tokens and effectively hiding the behavioral characteristics of the attack from static analysis.

4. **Repository Name:** The repository created to exfiltrate stolen data uses a name of eighteen random characters, with the fixed description "Sha1-Hulud: The Second Coming", confirming its role as the successor to the `Shai-Hulud` 1.0 attack.

5. **Destructive Behavior:** Finally, if the malware fails to exfiltrate data or propagate itself, for example, because no valid credentials are found, it attempts to permanently delete the home directory of the user, destroying all local files.

Taken together, these changes made `Shai-Hulud` 2.0 significantly more dangerous than its predecessor, as it spread faster, reached over a thousand packages within hours, executed its payload more rapidly thanks to the Bun runtime, introduced destructive capabilities absent in the original version, and its heavily obfuscated code made static analysis considerably harder.

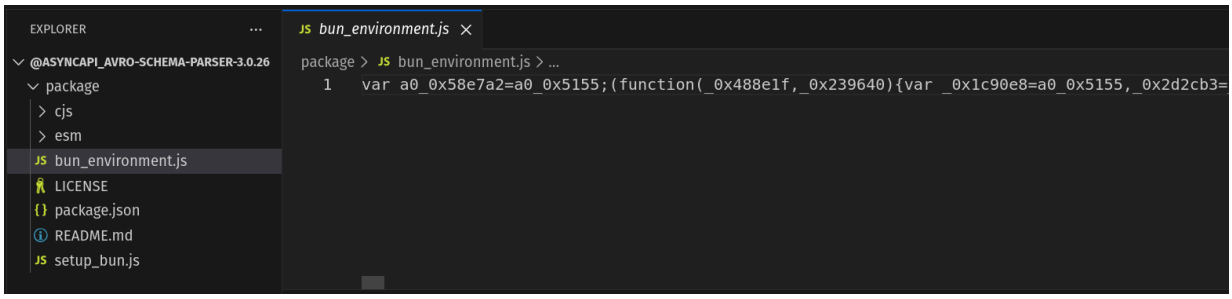

Figure 4.5: The `bun_environment.js` File of the Compromised `@asyncapi/avro-schema-parser` Package in Attack D.

Chapter 5

Experimental Setup

This chapter describes the datasets used for the analyses conducted in Chapter 6 and the software tools developed to create them. Section 5.1 introduces the two datasets, the goodware dataset and the malware one, explaining why they were structured this way, how the entries were selected, and what their main statistical properties are. Section 5.2 details the two developed software tools: the primary analysis application that downloads tarballs and computes metrics on them, and the custom script to automatically retrieve malicious versions.

5.1 Datasets

To carry out the analyses in Chapter 6, two datasets are required: one to establish a baseline of normal package behavior, and one to evaluate whether malicious releases are detectable against it. Accordingly, we constructed a *goodware* (Section 5.1.1) and a *malware* dataset (Section 5.1.2). Both consist of the same set of metrics computed from npm tarballs, using the tool described in Section 5.2.1. Concretely, for each package across its 20 latest releases, these datasets store: the set of distinct file types detected, the total uncompressed size in bytes, and, in plain-text files, the length of the longest line, and the count of unique hexadecimal values and Ethereum addresses. While the metrics themselves are analyzed in Chapter 6, this section focuses on the composition and construction of both collections.

Each dataset is organized as a collection of CSV files, one per metric. Each is named after the metric it represents, followed by a suffix indicating the dataset: `_GD` for the goodware and `_MD` for the malware (e.g., `package_size_GD.csv`, `package_size_MD.csv`). As illustrated in the excerpts provided in Tables 5.1 and 5.2, the first column provides the entry name, followed by 20 columns, labelled -19 to 0, that hold the metric value

for the corresponding release. For the *Unique Hexadecimal Values* and *Unique Ethereum Addresses* metrics, an additional filtered variant of the goodware CSV is also provided, containing only the entries that have at least one non-zero value in any release (e.g., `unique_hexadecimal_values_GD_GreaterThan0.csv`). This filtered file is used in Chapter 6 to focus the analysis on modules that are actually informative for those metrics. The CSV format was chosen for three main reasons: it is human-readable without specialized software, it integrates naturally with data analysis libraries such as `pandas` [Teac] and `matplotlib` [Teab], and it enables independent reuse of the datasets without requiring any specific environment. Both datasets are publicly available on GitHub[1].

A key design decision shared by both is the fixed window of exactly 20 releases per entry. This constraint ensures comparability across packages and between datasets, and reflects the deliberate methodological choice of tracking the evolution of each across consecutive releases rather than analyzing a single snapshot in isolation. The threshold of 20 was chosen to balance two competing requirements, as it is sufficiently large to capture a meaningful historical trend while remaining small enough to retain a large fraction of the ecosystem. A higher threshold would have excluded significantly more packages from the goodware dataset. This longitudinal perspective is essential to the detection approach, because malicious updates stand out as sudden deviations from an otherwise predictable evolutionary pattern, a signal that would be invisible in a single-snapshot analysis. However, this represents a trade-off and a limitation, as for long-lived projects with many releases, examining the 20 most recent versions may capture a partial view of their history, omitting behavioral patterns present in the early stages of development.

5.1.1 Goodware Dataset

The purpose of the goodware dataset is to establish a *baseline*, the typical behavior of each metric under normal, benign development conditions. This baseline is used in Chapter 6 to identify packages exhibiting outlier values and to evaluate whether the compromised releases in the malware dataset are detectable.

To build a large and representative baseline, the dataset is composed of modules ranked among the 1,000 most popular in the npm registry by number of stars, according to a publicly available popularity ranking [Tri26]. This criterion was chosen because such packages are the most realistic and impactful targets of SSCAs. Of the 1,000 candidates, 371 have fewer than 20 releases available in the registry and were therefore excluded, as the analysis requires exactly that number per entry. The dataset includes the remaining 629 packages, using the 20 most recent versions parsable via node-semver [Pre] (ensuring chronological order). Within the CSV file, entries are sorted by descending order of popularity, so that

[1] https://github.com/Frazzerz/Datasets-npm-Analysis

the most downloaded modules appear first. The developed tool (Section 5.2.1) downloads the corresponding tarballs automatically, taking as input a json file containing the 1,000 names extracted from the same ranking [Tri26].

Table 5.1: Excerpt of `package_size_GD.csv` (values in bytes).

	Version							
Package	−19	−18	−17	−16	...	−3	−2	−1	0
`chalk`	26,924	31,421	31,889	32,729	...	44,242	44,295	44,342	44,342
`commander`	162,734	163,823	168,750	169,596	...	200,514	208,171	208,150	208,153
`debug`	79,494	79,525	79,525	53,255	...	42,060	42,798	42,793	42,793
`tslib`	32,856	34,201	34,220	35,212	...	84,876	86,155	89,090	90,359

Table 5.1 shows an excerpt of `package_size_GD.csv`, illustrating two typical evolutionary patterns. `commander` and `tslib` show a gradual size increase across versions, reflecting the natural accumulation of functionality over time; by contrast, `debug` decreases significantly from the middle of its history due to a major restructuring that made it considerably lighter. These variations are expected and follow predictable, documented trajectories, a property that makes them distinguishable from the abrupt spikes introduced by compromised releases, as shown in Table 5.2. It is also worth noting that for `chalk` and `debug`, the value at version 0 is identical to version -1, and this reflects the standard maintainer response of unpublishing the malicious version and immediately republishing the last clean one, as described in Section 2.2.

Table 5.2: Excerpt of `package_size_MD.csv` (values in bytes). The malicious version of each package is highlighted in bold.

	Version						
Package	−19	−18	−17	**...**	−2	−1	0
`eslint-config-prettier`	18,332	19,729	19,819	...	59,021	**1,356,692**	59,021
`chalk`	31,421	31,889	32,729	...	44,342	**121,131**	44,342
`@ctrl/tinycolor`	238,949	288,469	241,608	...	244,065	**3,978,776**	244,090
`@asyncapi/bundler`	56,101	56,128	56,473	...	64,914	64,883	**10,227,467**

The 629 packages in the dataset span a wide range of sizes and complexities, reflecting the diversity of the real-world npm ecosystem. Table 5.3 summarizes the key statistical properties of the goodware dataset for each of the five metrics analyzed in Chapter 6. For each, the table reports the median and maximum values, the number of packages that have at least one non-zero value across their versions (Packages > 0), and representative notable examples.

For *File Types*, the median and maximum refer to the number of unique file types found per release. The maximum of 27 is attributable to `moment-timezone` (in version 0.5.36), which

erroneously included an unrelated temporary folder, subsequently removed in the following version, as discussed in Section 6.2. In the Notable examples column, the three most common extensions are reported, together with the percentage of packages that contain at least one file of that type.

For *Package Size*, the median grows progressively over time, from approximately 71 KB in older releases to 84 KB in the most recent ones. The maximum of 211 MB corresponds again to `moment-timezone`, for the same reason described above and also detailed in Section 6.3. Other packages with notably large sizes include `next` and `aws-sdk`, which are consistently among the largest across all their versions.

The high variability in *Longest Line Length* is attributable to the presence of build artifacts in some tarballs. The maximum of 4,041,545 characters belongs to `next`, for this reason, similarly to those under the `@angular` scope, as detailed in Section 6.4.

For *Unique Hexadecimal Values*, 109 out of 629 packages (17.3%) have at least one hexadecimal value in any version; the remaining 520 have none. Among those with non-zero values, the ones with the highest counts are typically encryption-related, such as `crypto-js` and `node-forge`, or contain generated JavaScript files, such as `aws-sdk`. The maximum of 2,208 belongs to `polished`, which temporarily included an erroneous `.yarn` cache directory with a generated file, as detailed in Section 6.5.

For *Unique Ethereum Addresses*, only two packages (`brace-expansion` and `dotenv-expand`) contain a single address each, across their versions. In both cases, the address originates from a `tea.yaml` protocol registration file unrelated to the core functionality of either module, as discussed in Section 6.6.

The main limitation of this dataset is that it represents only popular packages; less prominent ones may exhibit different statistical properties, and the baselines derived here should be interpreted accordingly.

Table 5.3: Summary Statistics of the Goodware Dataset.

Metric	**Median**	**Max**	**Packages > 0**	**Notable examples**
File Types	5	27	629 (100%)	`json` (100%), `js` (∼97.6%) and `ts` (∼71%)
Package Size	∼84 KB	∼211 MB	629 (100%)	`next` (∼140MB) and `aws-sdk` (∼98MB)
Longest Line Length	∼460	4,041,545	629 (100%)	`next` and `@angular` (maps)
Unique Hexadecimal Values	0	2,208	109 (17.3%)	`crypto-js`: 1,588, `chardet`: 972, `aws-sdk`: 600
Unique Ethereum Addresses	0	1	2 (0.3%)	`brace-expansion` and `dotenv-expand` (`tea.yaml`)

5.1.2 Malware Dataset

The malware dataset contains packages compromised by one of the four SSCAs analyzed in Chapter 4, and is used in Chapter 6 to evaluate whether the metrics are sensitive enough to detect the anomalous malicious releases. The dataset includes twenty entries, each analyzed across exactly 20 releases: nineteen legitimate and one compromised. Within the CSV files, packages are ordered chronologically by attack: the three from Attack A appear first, followed by the nine from Attack B, the four from Attack C, and the four from Attack D. The nineteen legitimate versions were downloaded automatically by the tool described in Section 5.2.1, while the malicious tarballs were retrieved separately using the custom script presented in Section 5.2.2.

The twenty packages were selected based on two criteria. The first was the availability of at least nineteen legitimate prior versions in the registry, as required by the longitudinal analysis design. The second was a sufficiently high weekly download count to reflect realistic, high-impact attack scenarios. Entries that did not satisfy the first criterion were therefore excluded; specifically, two from Attack A and nine from Attack B lacked enough prior versions. For Attacks C and D, which affected a very large number of packages (187 and 1,055, respectively), including all compromised ones would not add analytical value and would make the dataset difficult to manage; therefore, only a representative subset was included, selecting those with at least 1,000 weekly downloads. It is worth noting that all nine packages included from Attack B are among the 1,000 most popular in the registry, while none of those from the remaining attacks fall into that category.

The full list of entries included in the malware dataset, together with the specific compromised release number for each, is reported below.

- Attack A (Section 4.2):
  - `eslint-config-prettier`, v.10.1.7
  - `eslint-plugin-prettier`, v.4.2.3
  - `synckit`, v.0.11.9
- Attack B (Section 4.3):
  - `ansi-regex`, v.6.2.1
  - `ansi-styles`, v.6.2.2
  - `chalk`, v.5.6.1
  - `color-convert`, v.3.1.1
  - `color-string`, v.2.1.1

  - `debug`, v.4.4.2
  - `supports-color`, v.10.2.1
  - `strip-ansi`, v.7.1.1
  - `wrap-ansi`, v.9.0.1
- Attack C (Section 4.4):
  - `@ctrl/deluge`, v.7.2.1
  - `@ctrl/shared-torrent`, v.6.3.2
  - `@ctrl/tinycolor`, v.4.1.2
  - `ember-browser-services`, v.5.0.3
- Attack D (Section 4.5):
  - `@asyncapi/avro-schema-parser`, v.3.0.26
  - `@asyncapi/bundler`, v.0.6.6
  - `@asyncapi/cli`, v.4.1.3
  - `posthog-node`, v.5.11.3

Table 5.2 presents an excerpt of `package_size_MD.csv` with one entry per attack, illustrating how the malicious version appears in the malware dataset. As observed in Table 5.2, for the *Package Size* metric, the compromised release in all SSCAs introduces a sharp, isolated spike surrounded by otherwise predictable evolutionary patterns in the legitimate values. For `eslint-config-prettier` and `chalk`, the value at version 0 returns to the pre-attack level because the maintainer republished the last clean release, as discussed in Section 5.1.1. Instead, for `@asyncapi/bundler`, the spike falls at version 0, because the PM chose to downgrade to the prior version rather than republishing it as a new one, as described in Section 2.2.

The main limitation of this dataset is that it covers a narrow selection of attack campaigns; packages compromised by different SSCAs may exhibit structural properties not captured by this sample, and the results should be interpreted accordingly.

5.2 Software Tools

This section describes the tool and script developed to create the two datasets presented in Section 5.1. Section 5.2.1 presents the primary analysis application, namely `npm Packages Analyzer`, while Section 5.2.2 details the custom script to automatically retrieve malicious versions from mirror registries.

5.2.1 The npm Packages Analyzer

The developed tool computes the metrics analyzed in Chapter 6 from a set of npm package releases. It enables the construction of both the goodware (Section 5.1.1) and the malware dataset (Section 5.1.2). Implemented in Python, it is available on GitHub[2], and can be invoked from the command line as follows:

```
python3 main.py --json <input_file>.json [--workers N] [--local]
```

The required input is a json file containing the list of npm package names to analyze. The optional `--local` flag allows local tarballs, such as the malicious releases in the malware dataset, to be included alongside those downloaded from the registry. The optional `--workers` flag controls the number of parallel processes used for per-file metric computation. Packages and their versions are processed sequentially (from oldest to most recent), while the analysis of individual files within each version is parallelized using multiprocessing. The tool workflow is organized into four phases, described below.

5.2.1.1 Phase 1: Version Retrieval

The `npmClient` component queries the official npm registry to retrieve the module metadata and the list of all available releases. From this list, only versions parsable by node-semver [Pre] are kept, so that they can be sorted chronologically. For the goodware dataset (Section 5.1.1), the 20 most recent valid versions are selected and their tarballs downloaded. For the malware dataset (Section 5.1.2), only the nineteen most recent legitimate versions are fetched; the twentieth slot is reserved for the malicious tarball, which is retrieved separately via the custom script (Section 5.2.2) and included through the `--local` flag.

5.2.1.2 Phase 2: Metrics Computation

For each version, the tool first extracts the tarball and computes the metrics on the individual files it contains.

The *File Types* metric (Section 6.2) is computed by running `Magika` [Goo], a deep-learning-based content-type detection library developed by Google. Each file in the tarball is processed, and the resulting label is recorded. Notably, `Magika` labels Windows Portable Executable (PE) files (including DLLs) as `pebin`. The *Package Size* metric (Section 6.3) is computed as the uncompressed size in bytes of each individual file.

[2] https://github.com/Frazzerz/npm-packages-analyzer

Metrics based on the textual content of files, namely *Longest Line Length* (Section 6.4), *Unique Hexadecimal Values* (Section 6.5), and *Unique Ethereum Addresses* (Section 6.6), are computed only on plain-text sources, such as JavaScript and TypeScript, and are therefore skipped for binary formats such as zip, png or DLL. To determine the nature of each item and thereby distinguish plain-text from binary content, the tool uses `Magika` to classify each entry and checks the detected label against a predefined whitelist of allowed types. Additionally, for these metrics, comments are stripped from JavaScript and TypeScript code before computation, using a parser from the `tree-sitter` library [Bru], so that comment content does not affect the results. To compute *Longest Line Length*, the tool splits the buffer at newline characters and records the length of the longest line.

The *Unique Hexadecimal Values* and *Unique Ethereum Addresses* metrics are computed via regular expressions (regex). These are precompiled with `re.compile` from the Python `re` library [Fou], which parses each pattern once and reuses the resulting object across all files, reducing overhead. The precompilation step also allows flags to be specified at compile time; for example, `re.IGNORECASE` is used to match both `0x` and `0X` prefixes in the hexadecimal logic, making the search case-insensitive. To find all matches within a file contents, the tool uses the `finditer` method on the compiled object, which returns an iterator over all matches and is more memory-efficient than collecting all results at once.

The regex used for *Unique Hexadecimal Values* (Section 6.5) is: `_?0x[0-9a-fA-F]+`

The optional leading underscore (specified with `_?`) captures the naming convention used by obfuscation tools when a hexadecimal token serves as a variable or function name (e.g., `_0x6fd57a`), followed by the mandatory `0x` prefix with one or more hexadecimal characters (digits 0 to 9 and letters a to f, both lowercase and uppercase). This flexible matching rule allows capturing values even when they appear inside function calls or other code structures, where strict boundaries, would miss them.

For *Unique Ethereum Addresses* (Section 6.6), the pattern is instead:
`(?<![\w\-/_])0x[a-fA-F0-9]{40}(?![\w\-/_])`

A valid Ethereum address consists of the prefix `0x` followed by exactly 40 hexadecimal characters [BC14]. The lookbehind (`(?<![\w\-/_])`) and lookahead (`(?![\w\-/_])`) prevent matching strings embedded in URLs, file paths, or longer hexadecimal sequences, which standard word boundaries like `\b` would not correctly handle, given the presence of typical path separators such as slashes and hyphens.

Once per-file metrics are available, they are aggregated to produce a single version-level summary. For *File Types*, the per-file labels are collected into a deduplicated set of unique types. For *Package Size*, the per-file sizes are summed. For *Longest Line Length*, the maximum across all plain-text files is taken. For *Unique Hexadecimal Values* and *Unique Ethereum Addresses*, the per-file match lists are merged and deduplicated.

5.2.1.3 Phase 3: Results Saving

For each package, results are saved incrementally to two CSV files as each version is processed, so that partial outputs are preserved in the event of an interruption or error. `file_metrics.csv` records one row per file across all analyzed releases, with columns for each per-file metric, as illustrated in Table 5.4, while `aggregate_metrics_by_single_version.csv` records one row per release, with columns for the corresponding version-level aggregated values, as shown in Table 5.5. Furthermore, a `log.txt` file is saved, updated incrementally, mirroring the terminal output. It reports general information such as the total number of packages and worker count, and for each entry it logs the number of versions found, the download status of each tarball, the per-release analysis progress, and the total analysis time.

Table 5.4: Excerpt of `file_metrics.csv` for `chalk` package.

Version	File Path	File Type	Size Bytes	Longest Line	Hex Count	Hex Patterns	...
2.4.2	`package.json`	json	1,195	74	0	[]	...
2.4.2	`index.js`	javascript	6,439	120	0	[]	...
2.4.2	`types/index.d.ts`	typescript	2,353	65	0	[]	...
2.4.2	`templates.js`	javascript	3,133	158	0	[]	...
...	...	...	...	...	...	...	
3.0.0-beta.1	`package.json`	json	1,003	53	0	[]	...
3.0.0-beta.1	`source/index.js`	javascript	5,863	135	0	[]	...
3.0.0-beta.1	`source/templates.js`	javascript	3,361	179	0	[]	...
...	...	...	...	...	...	...	
5.6.2	`package.json`	json	1,640	59	0	[]	...
5.6.2	`source/index.js`	javascript	5,902	140	0	[]	...
5.6.2	`source/.../index.js`	javascript	5,256	109	1	[“0xFF”]	...

Table 5.5: Excerpt of `aggregate_metrics_by_single_version.csv` for `chalk` package.

Version	List Total Types	Total Types	T. Size Bytes	Longest Line	T. Hex	Hex Patterns	T. Eth. Add.	Eth. Add. Patterns
2.4.2	[“json”, ...]	5	26,924	158	0	[]	0	[]
3.0.0-beta.1	[“json”, ...]	5	31,421	179	0	[]	0	[]
...	...	...	...	...	...	...	...	...
5.6.2	[“json”, ...]	5	44,342	140	1	[“0xFF”]	0	[]

5.2.1.4 Phase 4: Dataset Generation

Once all packages have been analyzed, the dataset files are produced by running:

```
python3 dataset.py
```

This script reads the `aggregate_metrics_by_single_version.csv` file from every analyzed package and generates a separate CSV file per metric, where each row corresponds to an entry and each column to one of its analyzed versions, as illustrated in Tables 5.1 and 5.2. Each resulting file contains the metric values for every version of every analyzed package, collectively forming the datasets described in Section 5.1.

5.2.2 Download Malicious Tarballs

This section describes the process and the developed script used to retrieve the tarballs of the compromised releases specified in Section 5.1.2.

The first approach was to search public datasets that aggregate known malicious versions from open-source registries [Dat23]. The limitation of such repositories is that they do not always contain the specific releases of interest: in our case, the compromised versions of the packages affected by the four SSCAs analyzed in Chapter 4 were not initially present in these collections.

An alternative source of removed npm tarballs, which proved effective, is to query mirror registries. A mirror registry is a server that replicates the contents of an official package repository and periodically synchronizes with it [Guo+23]; examples of npm mirrors include *Huaweicloud*, *Taobao*, and *Tencent*. Because synchronization with the official registry is not instantaneous, mirrors often retain copies of releases that have already been unpublished upstream, sometimes for extended periods depending on the mirror synchronization frequency. This latency is an advantage in the context of malware research, as it provides a window during which a removed malicious tarball can still be retrieved.

To automate the retrieval of compromised tarballs across multiple mirrors, we developed a dedicated script, implemented in Python and publicly available on GitHub[3]. The script is invoked from the command line as follows:

```
python3 downloadpkgnpm.py <input_file>.json
```

The input is a json file specifying the list of mirrors to query and the list of package-version pairs to download. For each requested release, the script queries the mirrors sequentially until the tarball is found and downloaded; if all mirrors fail, the failure is reported and the script moves on to the next entry. Once retrieved, the tarballs can be passed to the `npm Packages Analyzer` tool (Section 5.2.1) via the `--local` flag to construct the malware dataset (Section 5.1.2).

[3] https://github.com/Frazzerz/Download-malicious-npm-packages

Chapter 6

Metrics Analysis

This chapter motivates, analyzes, and evaluates the five metrics used in this study to detect SSCAs discussed in Chapter 4. Section 6.1 describes the methodology, how each metric is analyzed, how the baseline is established, and how to read the figures. Section 6.2 through Section 6.6 each cover one metric in full, while Section 6.7 synthesizes the results and discusses the overall effectiveness, limitations, and cost of the proposed approach.

6.1 Methodology

In this context, a metric is a measure computed from a specific version of an npm package. While the literature offers a broad range of indicators based on code analysis [Zor25; Wei+25], we focused on a set of metrics defined ad hoc, derived from the most relevant characteristics of the four SSCAs selected as case studies. Metrics unrelated to these attacks, such as the total number of files in the tarball or API-call patterns hidden behind obfuscation, were excluded. The five selected metrics are: *File Types* (Section 6.2), *Package Size* (Section 6.3), *Longest Line Length* (Section 6.4), *Unique Hexadecimal Values* (Section 6.5), and *Unique Ethereum Addresses* (Section 6.6).

Each metric is discussed in a dedicated section, presenting the motivation behind its choice and the two analyses. The first analysis characterizes the distribution and evolutionary patterns of the metric within the goodware dataset (Section 5.1.1), with the goal of establishing a baseline for normal development conditions. Understanding how a package evolves across versions is fundamental to evaluating potential attacks, and tracking a metric across consecutive releases allows us to identify anomalies that would be invisible in a single-snapshot analysis. A sudden deviation from an otherwise stable trend is precisely the signature left by the SSCAs analyzed in Chapter 4. Moreover, this longitudinal per-

spective helps distinguish malicious injections from legitimate development events, such as a new feature introduction or a major refactoring, that can also produce outlier values but are traceable to documented development decisions.

Except for the *File Types* metric (Section 6.2), which requires specialized representations, all others share the same figure format for the goodware dataset analysis. A concrete example is Figure 6.3, which shows the evolution of the *Package Size* metric across 20 releases. The x-axis represents the versions from the least recent (-19) to the latest available (0), and the y-axis reports the measured value on a logarithmic scale. The vertical black bars mark the 95th-percentile confidence intervals, computed following an established statistical approach [Pat+25]: entries within the interval exhibit values consistent with normal behavior, while the gray dots above the bar represent the outliers of the remaining 5% of the population. The red dashed line is the median, calculated for each release and used instead of the mean to avoid distortion from extreme values. Packages of particular interest are highlighted with colored lines, as illustrated in Figures 6.4 and 6.6. For instance, Figure 6.4 (discussed in Section 6.3) shows `moment-timezone` (solid green) exhibiting a sudden jump at ②, caused by the accidental inclusion of a temporary folder. Instead, in Figure 6.6 `next` (solid orange) remains a stable outlier across all releases due to the large build artifacts it bundles by design. Comparing these two cases already illustrates the core challenge of this analysis, as both produce outlier values, but for entirely different reasons; distinguishing a legitimate anomaly from a malicious injection requires examining the nature and history of the change.

The second analysis evaluates the effectiveness of each metric by comparing the values of the compromised releases in the malware dataset (Section 5.1.2) against the goodware baseline. A metric is considered *effective* for a given attack if the compromised release produces a value that falls among the outliers of the goodware distribution, making it distinguishable from the majority of the population. Conversely, a metric is considered *not indicative* if the malicious release falls within the confidence interval, meaning it is indistinguishable from the majority of legitimate releases. Except for the *File Types* metric, all others share the same figure format for this comparison. Two concrete examples are Figures 6.8 and 6.10, both for the *Package Size* metric. In both figures, the colored lines show the longitudinal evolution of the affected entries, providing the context needed to appreciate how sudden the anomaly introduced by the malicious release actually is. Figure 6.10 shows an effective case, where the red crosses marking the compromised releases of Attack D fall among the outlier dots of the goodware distribution, confirming that the metric successfully detects this attack. Instead, Figure 6.8 shows a not indicative case, where the red crosses remain within the confidence intervals, meaning that the affected releases of Attack B are indistinguishable from legitimate ones under this measure. When packages from the same attack overlap visually, separate figures are provided per entry for clarity, as shown in Figure 6.21.

6.2 Analysis Metric 1: File Types

The first metric we examined relates to the set of file types in an npm package version. It is driven by the characteristics of Attack A (Section 4.2), in which an adversary inserts a harmful binary Windows executable into the tarball. The intuition is that DLL files are essentially never present in legitimate npm packages; thus, their appearance in a new version constitutes a clear anomalous signal. More generally, the presence of any file type that is rare within the goodware dataset warrants further investigation. To verify this, we first identified which categories are typical or rare in the goodware dataset, and then checked whether the malicious versions deviate from that baseline distribution. As demonstrated in this section, the analysis confirmed that binary Windows executables are virtually absent from benign entries, making their detection a strong anomalous signal.

6.2.1 Analysis of the Goodware Dataset

The goodware dataset provides, for each of the 629 packages, the set of file types found across their 20 analyzed versions, as detailed in Section 5.1.1. In total, 71 distinct extensions were identified throughout the dataset.

To visualize the distribution and evolutionary patterns of these file types across releases, we use two heatmaps, presented in Figures 6.1 and 6.2. In both, the x-axis shows the 20 versions, from the least recent (-19) to the latest available (0), while the y-axis lists the file extensions. Each cell contains the *relative frequency* of that extension, defined as the fraction of packages in the dataset containing at least one file of that type in a given version. For instance, a value of 1.000 indicates that every project contains that file type, whereas 0.010 indicates that only 1% do. Dark shading corresponds to high frequency, while light tones indicate lower occurrence. The 71 identified types are split between the two heatmaps as follows:

1. **Common File Types** (Figure 6.1): represents the most common categories, with a relative frequency exceeding 1% in at least one version.
2. **Rare File Types** (Figure 6.2): represents rarer ones, include all remaining extensions whose frequency never surpasses the 1% threshold.

Figure 6.1 reveals several notable patterns in the evolution of common file types.

As expected, **json files** are present in all packages across all versions, because every valid npm package must include a `package.json` file as a condition of publication [npm26]. However, **JavaScript files** are not always included, and this can occur for several reasons:

1. *Maintainer Oversight*: packages such as `@opentelemetry/api` (version 1.0.0-rc.2), `cross-fetch` (3.2.0-alpha.0), and `cheerio` (1.0.0-rc.7), among others, were published without JavaScript files by mistake, as documented in their GitHub release notes [Mai], and the versions were subsequently deprecated. This explains the slight dips in JavaScript frequency visible in some version columns.

2. *Project Abandonment*: some packages are no longer maintained, and their latest versions contain no code, as with `@types/uuid`.

3. *Package Purpose*: some packages provide data rather than executable logic, and developers use them as dependencies to retrieve current data without manual hard-coding. For example, some packages contain only TypeScript code, such as those under the `@types` scope (e.g., `@types/node`, `@types/react`), which provide static type definitions [Def]. Similarly, `node-releases`, which catalogues a list of Node.js release versions [Tob], and `spdx-license-ids`, which provides the official Software Package Data Exchange license list [Wat]; distribute information solely as json files without any JavaScript code.

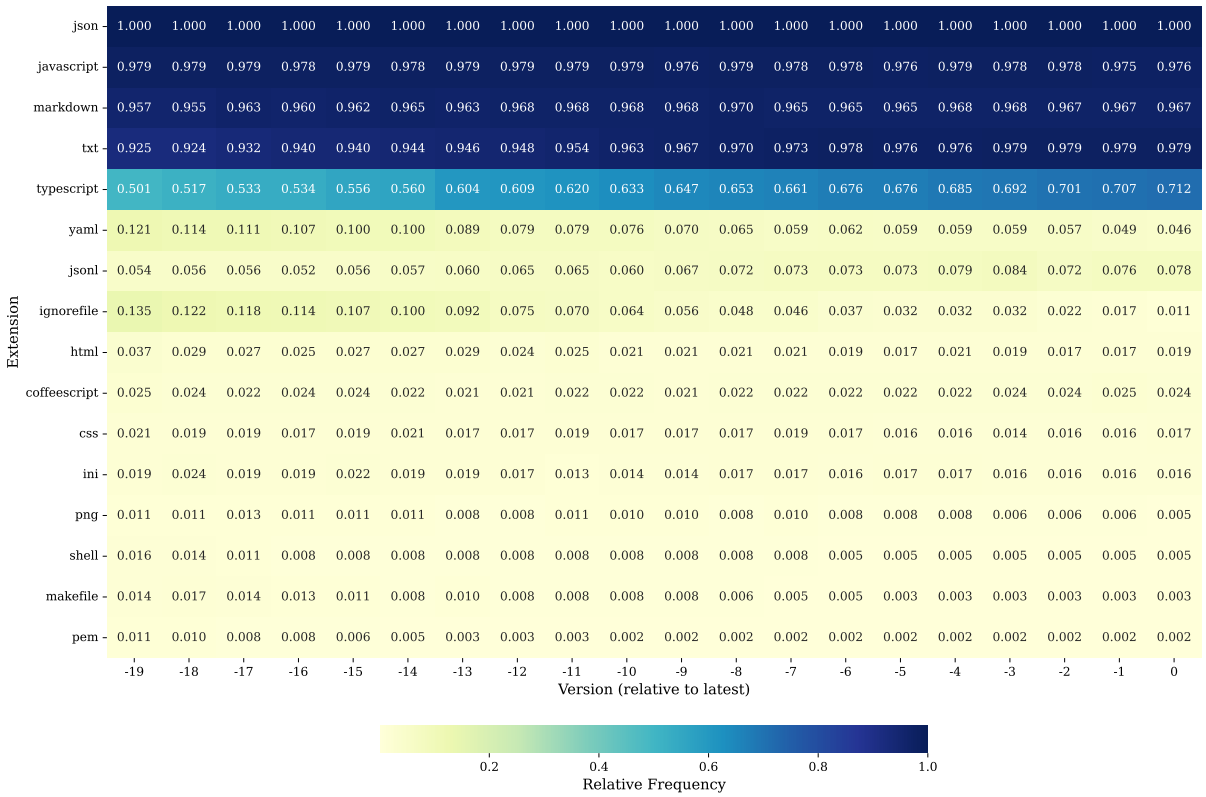

Figure 6.1: Heatmap for Common File Types.

The relative frequency of **TypeScript files** consistently increased across versions, rising from approximately 50% in the earliest releases to about 70% in the most recent. This reflects the broader ecosystem-wide shift toward native TypeScript support, driven by the benefits of static type checking for code quality and maintainability. This behavior is also visible in the size analysis of the `figlet` package (Section 6.3.1).

Markdown and text files, typically `README.md`, `CHANGELOG.md`, and `LICENSE`, are present in the majority of packages, with small variations attributable to occasional maintainer omissions. The remaining eleven file types in Figure 6.1 show declining relative frequency over time, never exceeding 8% in the latest versions, consistent with the practice of excluding non-essential files from tarballs to keep the package lightweight.

Shell files are present in a small and decreasing number of packages. In the most recent releases, three packages still include shell files: `pino`, which uses a script to update version numbers across its metadata files [CC]; and `husky` and `jake`, which use shell scripts to trigger automated checks at Git commit time [Typ; Eer]. These uses are legitimate and harmless.

Figure 6.2 displays the 55 file types whose frequency remains below 1% in any version. Four cases among these outlier occurrences deserve detailed attention.

C and C++ files are found in three packages: `sharp`, `node-addon-api`, and `nan`. `sharp` performs high-performance image processing using the C library `libvips` [Ful]. Since Node.js cannot execute C code natively, `sharp` depends on `node-addon-api` and `nan` [con; VB], which function as wrappers that bridge the JavaScript runtime and the compiled C code. The presence of C files in these packages is therefore an inherent requirement of their functionality.

gzip files appear in two packages as the result of accidental inclusions. `polished` mistakenly included a `.yarn` cache directory, containing `install-state.gz` among other files, in four consecutive versions (4.0.0-beta.2 to 4.0.3). This was corrected in release 4.0.4, as documented on GitHub [HS25]. Similarly, `moment-timezone` (0.5.36) accidentally included a temporary folder containing gzip, PE, mp3, and psd files (none related to its functionality), before removing it in the following version.

Dockerfile files appear in `recast` for three consecutive versions (0.21.3 to 0.21.5), due to the accidental inclusion of a `.devcontainer` folder intended for maintainer use only, which contains a dockerfile that configures a container with Node.js and TypeScript. This folder was removed in subsequent releases.

PE files (Windows executables) are present in two packages, marked as `pebin` in Figure 6.2. `nodemon` is a development tool that monitors file changes and automatically restarts the process [Sha], and includes `windows-kill.exe` in all 20 versions to handle signal events (such as CTRL+C) correctly on Windows (e.g., to allow clean process shut-

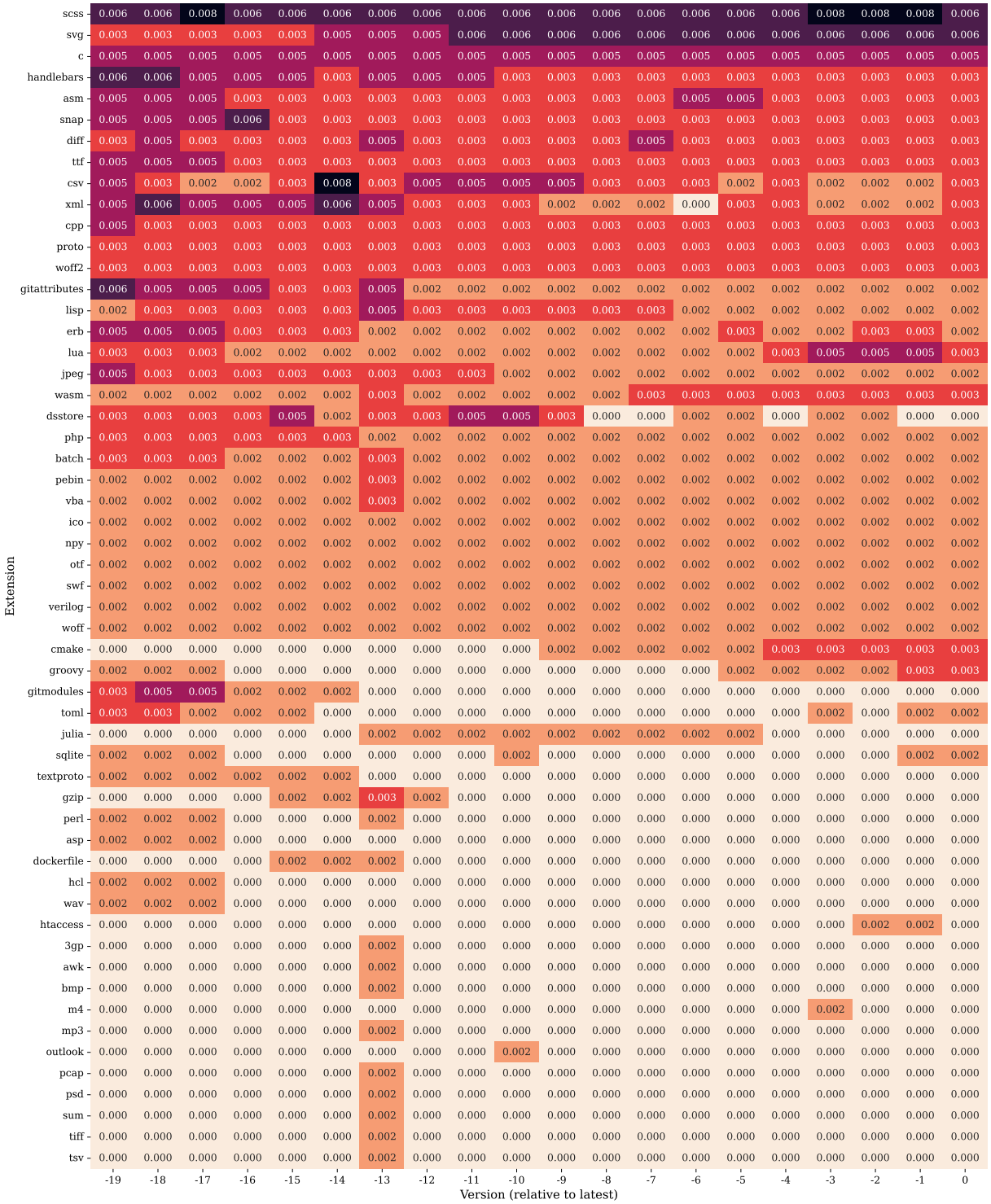

Figure 6.2: Heatmap for Rare File Types (Outliers).

down). Instead, `moment-timezone` includes a PE file by mistake in one version, as part of the temporary folder discussed above. The distinction between these two cases is important, as the `nodemon` PE file is stable, present in all releases, and has a documented purpose, whereas for the second package, an unexpected PE appears for the first time in a new update, without a documented justification. This second case should be treated as a high-risk anomaly, and follows a similar behavior pattern to Attack A.

6.2.2 Comparison with the Malware Dataset

In Attack A (Section 4.2), the adversary introduced a malicious Windows DLL into the tarball, and as shown in Figure 6.2, PE files are a rare extension in the goodware dataset, as only two packages contain them: `nodemon`, whose presence is documented and stable, and `moment-timezone`, where it was an accidental inclusion subsequently corrected. Therefore, the appearance of a PE file in a new version of a package that previously contained none is an outlier signal, making this metric effective for detecting Attack A.

For the remaining three attacks, the malicious payload either modifies an existing JavaScript file (Attack B) or introduces new ones (Attacks C and D), an extension present in nearly all packages in the goodware dataset. The metric is therefore not indicative of these SSCAs, since the compromised versions are indistinguishable from legitimate ones.

6.3 Analysis Metric 2: Package Size

The second metric is the uncompressed size of an npm package version, expressed in bytes. It is relevant because all four SSCAs introduce or modify files that increase the package size. Attack A (Section 4.2) incorporates a DLL of 1.2 MB, Attack B (Section 4.3) expands an existing JavaScript source code by adding a harmful string of 76,438 characters, Attack C (Section 4.4) bundles a file of approximately 3.7 MB, and Attack D (Section 4.5) injects a payload exceeding 10 MB. The question is whether these increments are significant enough to be distinguishable from the fluctuations that occur naturally during benign development.

Size is measured in absolute terms rather than as a relative change from the previous version. This choice is motivated by the heterogeneity of the npm ecosystem, as a small package may exhibit a large percentage increase even in response to a minor legitimate update, such as the addition of a single documentation file, whereas the same relative change would be negligible for a large package already weighing tens of megabytes. Absolute values therefore provide a more stable and comparable basis for establishing a baseline for the entire population.

6.3.1 Analysis of the Goodware Dataset

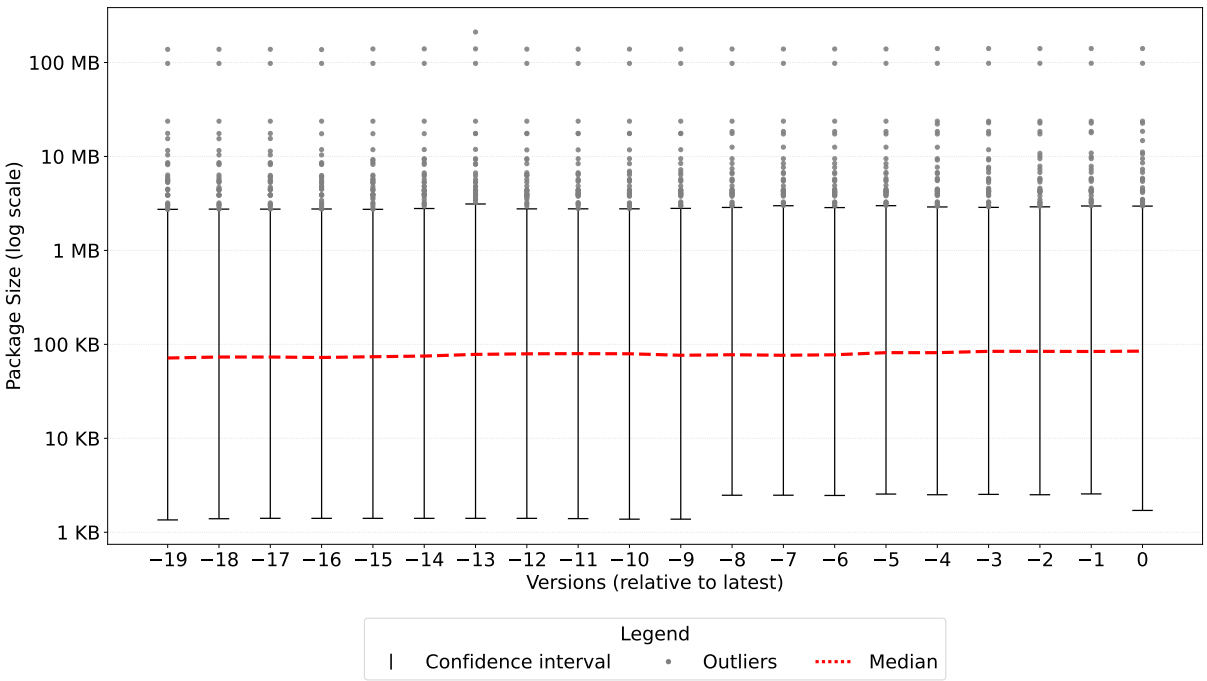

Figure 6.3: Package Size Metric: Package Trend in Goodware Dataset.

The goodware dataset provides, for each of the 629 packages, the total size observed in each of their 20 analyzed versions, as described in Section 5.1.1. To visualize the distribution and evolutionary patterns of this metric across releases, we used Figures 6.3 to 6.6. Among the packages in the goodware dataset, several exhibit outlier values for one or more versions. Nine packages were selected for individual analysis as representative cases, chosen to cover the range of observed trends and the most common causes of legitimate outliers, such as accidental inclusions, major refactorings, and inherent package characteristics. Understanding these patterns is essential to distinguish legitimate anomalies from malicious injections when evaluating the malware dataset in Section 6.3.2. The following figures highlight these nine packages with solid colored lines, categorizing them according to their observed trends:

- **Increasing Trend** (Figure 6.4): illustrates highlighted packages with an increasing trend over time.
- **Decreasing Trend** (Figure 6.5): shows highlighted packages characterized by a reduction in size.
- **Stable Trend** (Figure 6.6): displays highlighted packages maintaining a constant size throughout their history.

Figure 6.4 shows four packages that experienced a significant size increase across versions: `polished`, `moment-timezone`, `figlet`, and `date-fns`.

polished (solid pink line) is a package used for styling in JavaScript [HS]. As seen at ① in Figure 6.4, it grew from about 2.7 MB to 9.2 MB starting from version 4.0.0-beta.2 for four consecutive releases. In these versions, the developers erroneously included a `.yarn` cache directory in the final tarball distributed to the public, which contained numerous files, as also discussed in Section 6.2.1. The package returned to its previous size in version 4.0.4 once the folder was removed, as documented on GitHub [HS25].

As seen at ② in Figure 6.4, **moment-timezone** (solid green line) increased from about 1.6 MB to 211 MB in version 0.5.36 due to the unwanted inclusion of a temporary folder, making this the heaviest tarball in the entire goodware dataset. The folder was removed in the following release.

figlet (solid blue line) is used to create ASCII art text from normal strings, transforming letters into large stylized characters composed of symbols [Gil]. As seen at ③ in Figure 6.4, it grew from about 6.2 MB to 17.5 MB in version 1.9.0, when the project underwent a major refactoring to adopt TypeScript and modern build tooling, adding approximately 300 new files, as reported in the release notes on GitHub [Gil26]. This is an example of a legitimate, planned size increase with a clear documented cause.

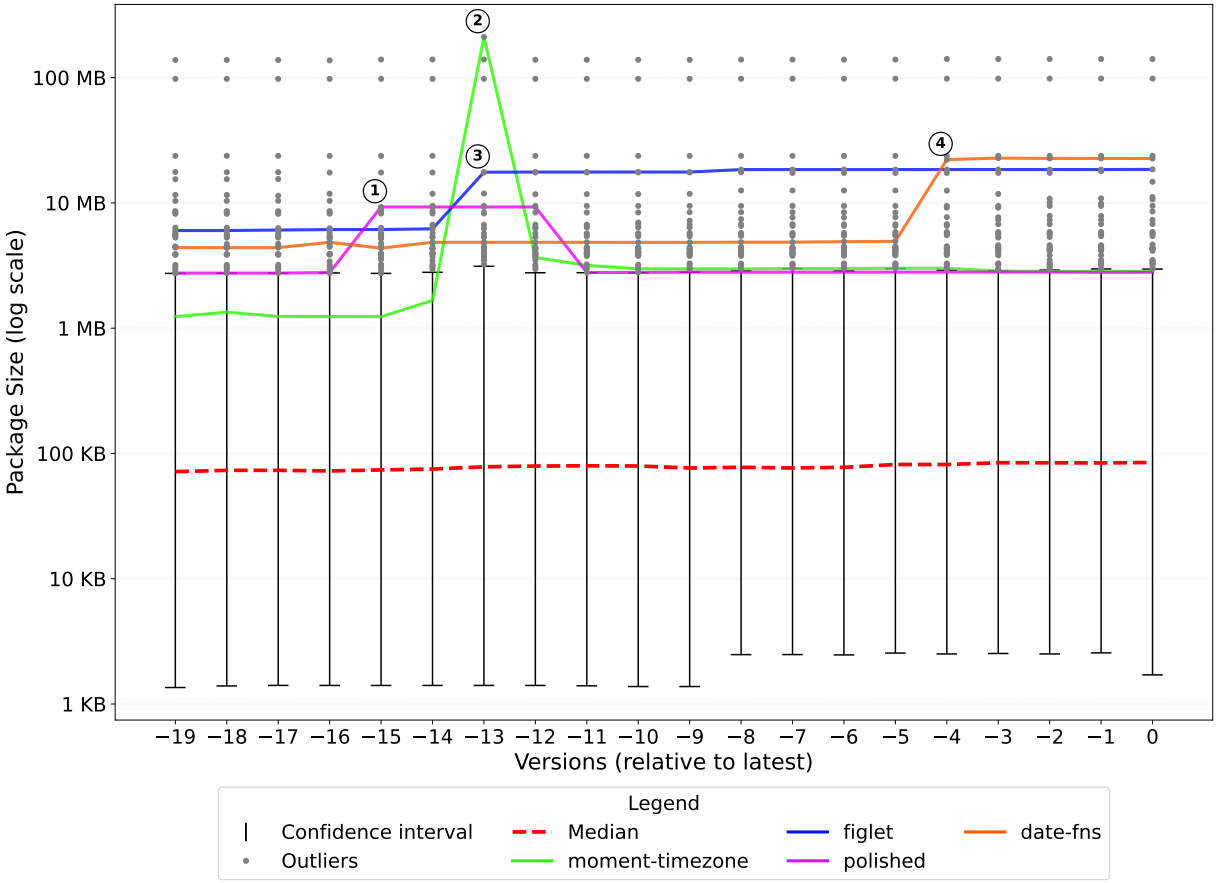

Figure 6.4: Package Size Metric: Outliers Package Increasing Trend in Goodware Dataset.

date-fns (solid orange line) provides a toolset for manipulating JavaScript dates in a browser and Node.js [tea]. As seen at ④ in Figure 6.4, it grew from about 5 MB to 22 MB in release 3.6.0, when the developers added 400 new files for compatibility with older browsers [tea26]. Again, this is a documented and legitimate change.

These examples illustrate how large size increases can occur in benign packages for a variety of reasons, such as unwanted inclusions, refactoring, or intentional feature additions.

Only the **punycode** package (solid green line in Figure 6.5) experienced a decrease large enough to move from an outlier to within the confidence interval between consecutive releases. In its first three versions, the size was 15.5 MB, and from ① in Figure 6.5, it was progressively optimized to include only the essential files, shrinking to approximately 33 KB in its most recent releases. This represents a deliberate and documented cleanup of the distributed tarball.

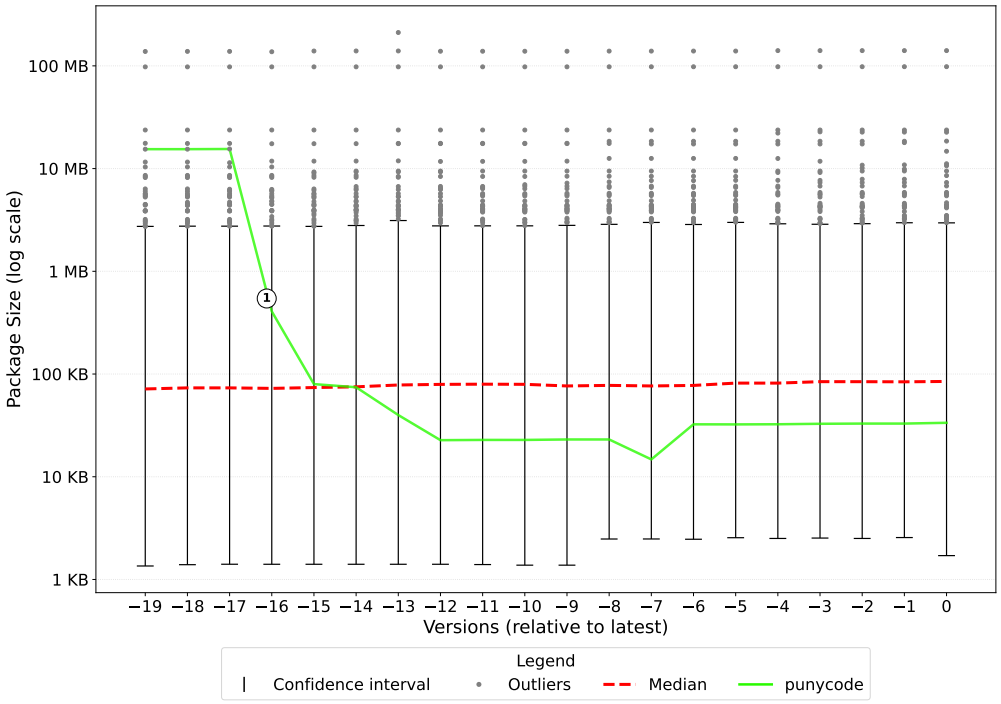

Figure 6.5: Package Size Metric: Outliers Package Decreasing Trend in Goodware Dataset.

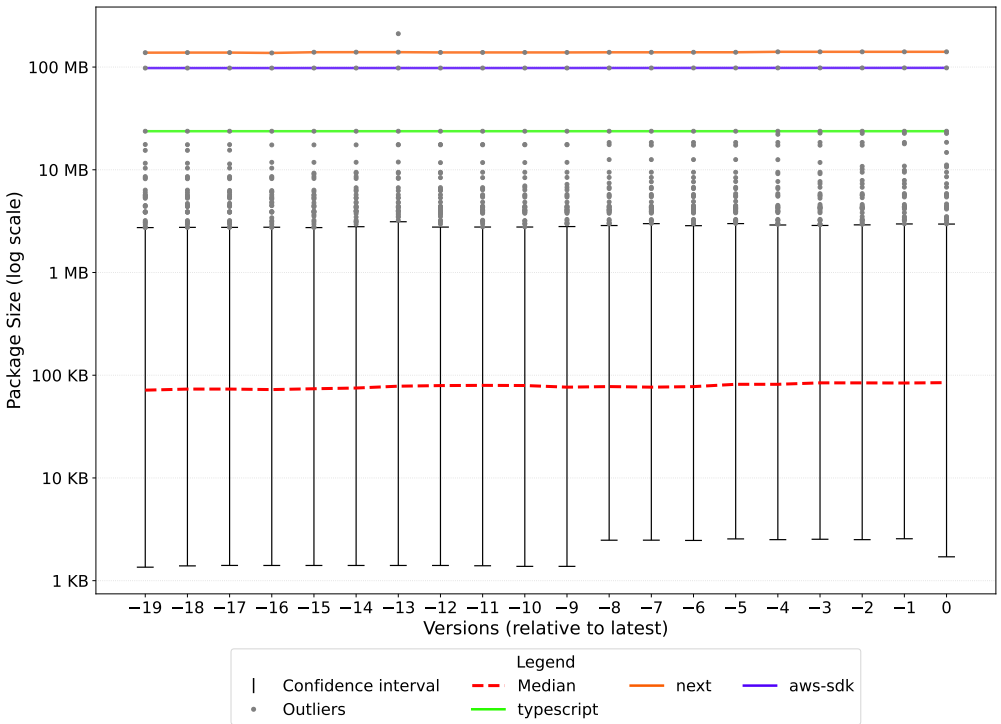

Figure 6.6: Package Size Metric: Outliers Package Stable Trend in Goodware Dataset.

As shown in Figure 6.6, three packages are consistently the largest in the dataset across all 20 versions: **next** [Ver] (solid orange, 140 MB, a React full-stack framework), **aws-sdk** [Ama] (solid purple, 98 MB, the official AWS client), and **typescript** [Mic] (solid green, 23.7 MB, the TypeScript compiler).

6.3.2 Comparison with the Malware Dataset

Attack A (Section 4.2) bundles a malicious DLL of 1.2 MB to each compromised package, and as shown in Figure 6.7, this increase is insufficient to place the compromised versions among the outlier values of the goodware dataset. The reason is that these affected tarballs are small enough to remain within the confidence intervals, failing to surpass the upper bound (approximately 2.7-2.9 MB) required for outlier classification. The metric is therefore not able to detect this specific attack and not indicative.

Attack B (Crypto attack, Section 4.3) adds a single obfuscated line to an existing `index.js` file. This increase is modest relative to the size range of the goodware dataset, and as shown in Figure 6.8, the compromised versions remain within the confidence interval, making the metric not indicative for this attack.

Attack C (`Shai-Hulud` 1.0, Section 4.4) injects a new file of approximately 3.7 MB. For the affected packages this addition is large enough to exceed the 95th percentile threshold, placing the compromised versions among the outliers, as shown in Figure 6.9. The metric is therefore effective for detecting Attack C.

Attack D (`Shai-Hulud` 2.0, Section 4.5) introduces a file of over 10 MB. As shown in Figure 6.10, this places the compromised versions well outside the outlier range, making the metric clearly effective for detecting Attack D.

In summary, the *Package Size* metric effectively detects the two attacks that introduce large malicious files (Attacks C and D), but is not sensitive enough to detect the smaller injections of Attacks A and B.

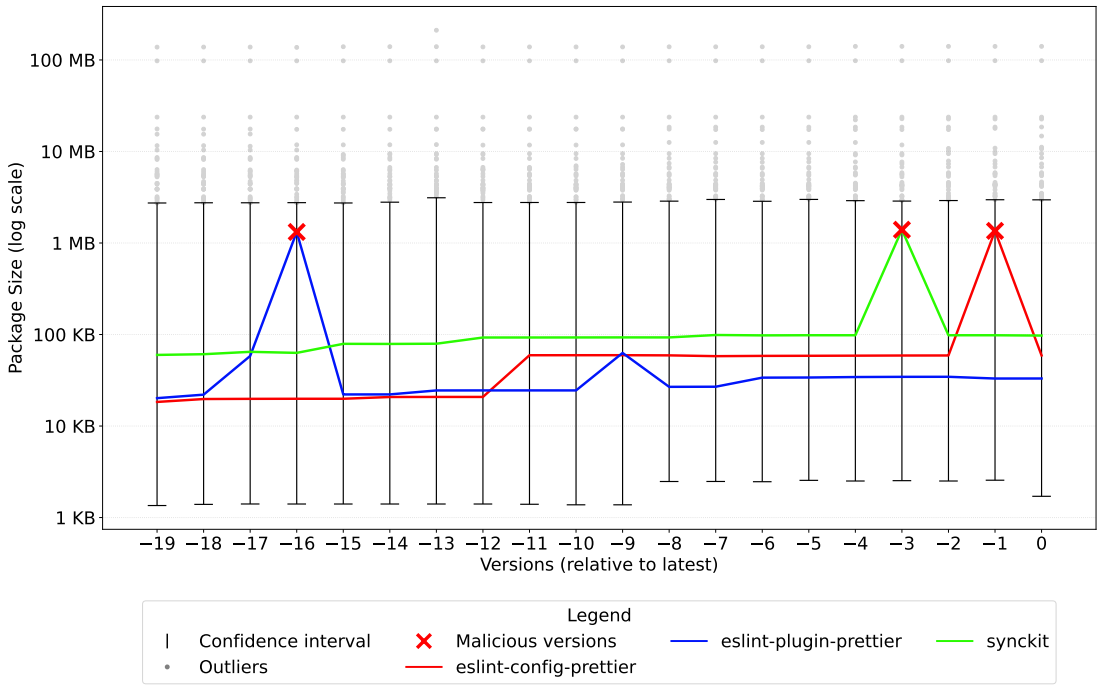

Figure 6.7: Package Size Metric: Malware Comparison for Attack A.

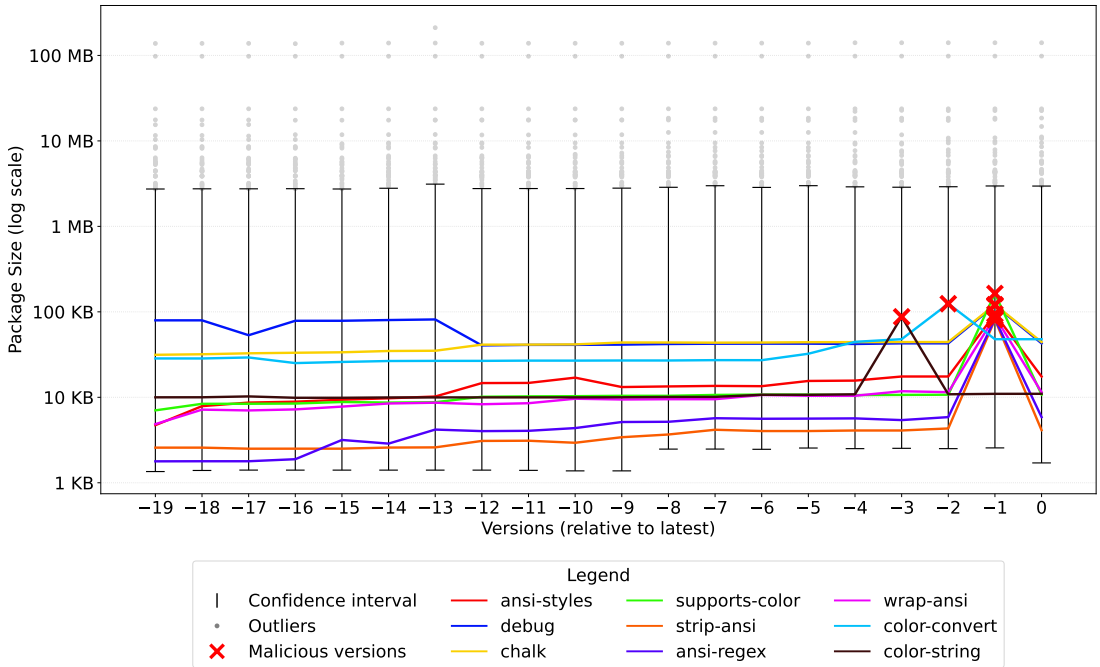

Figure 6.8: Package Size Metric: Malware Comparison for Attack B.

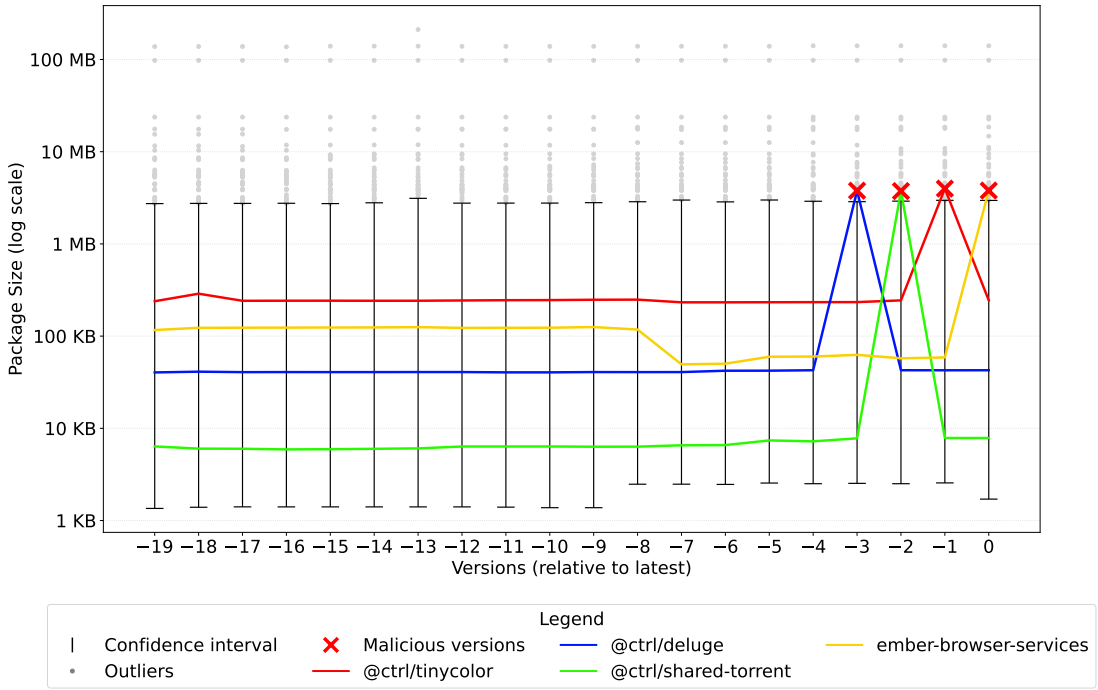

Figure 6.9: Package Size Metric: Malware Comparison for Attack C.

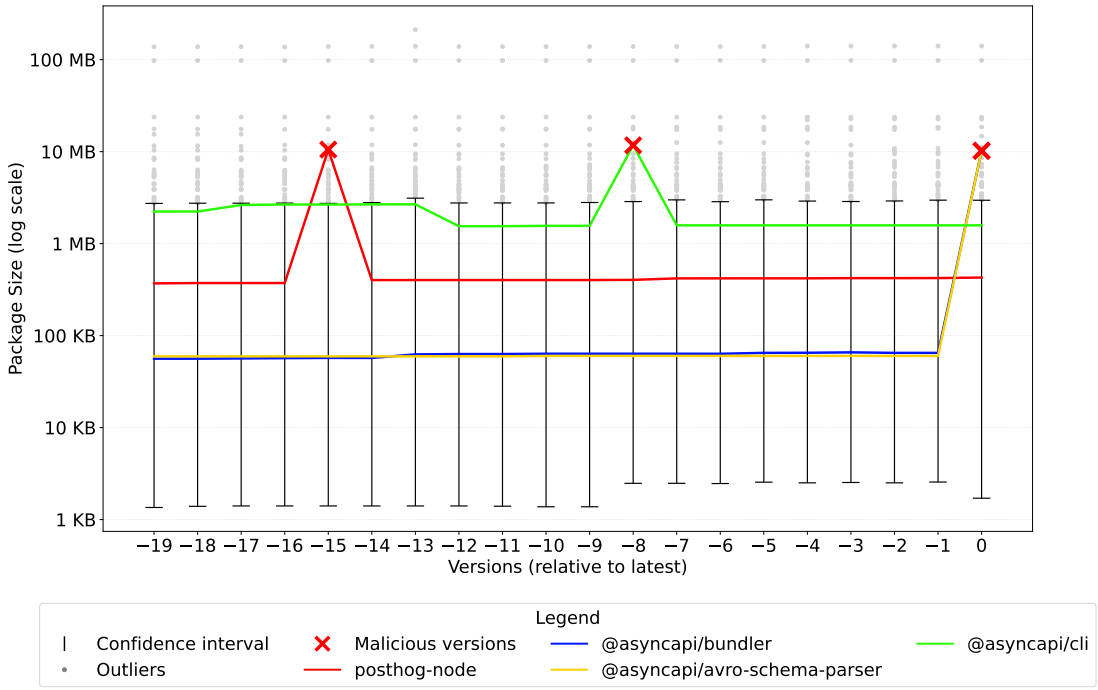

Figure 6.10: Package Size Metric: Malware Comparison for Attack D.

6.4 Analysis Metric 3: Longest Line Length

The third metric we analyzed is the length of the longest line found across all plain-text files of an npm package version, expressed in number of characters. It was motivated by the observation that Attacks B (Section 4.3), C (Section 4.4), and D (Section 4.5) all inject malicious code formatted as a single, extremely long line. Specifically, Attack B adds 76,438 characters to an existing `index.js`, while Attacks C and D introduce files containing lines of 3,734,355 and 10,157,373 characters, respectively. The intuition is that legitimate packages rarely contain code with lines of such extreme length, making this an anomalous signal. To verify this, we first characterized the distribution of this metric in the goodware dataset, paying particular attention to the role of build artifacts, automatically generated files that can also produce long lines, and then evaluated whether the malicious releases deviate detectably from that baseline.

6.4.1 Analysis of the Goodware Dataset

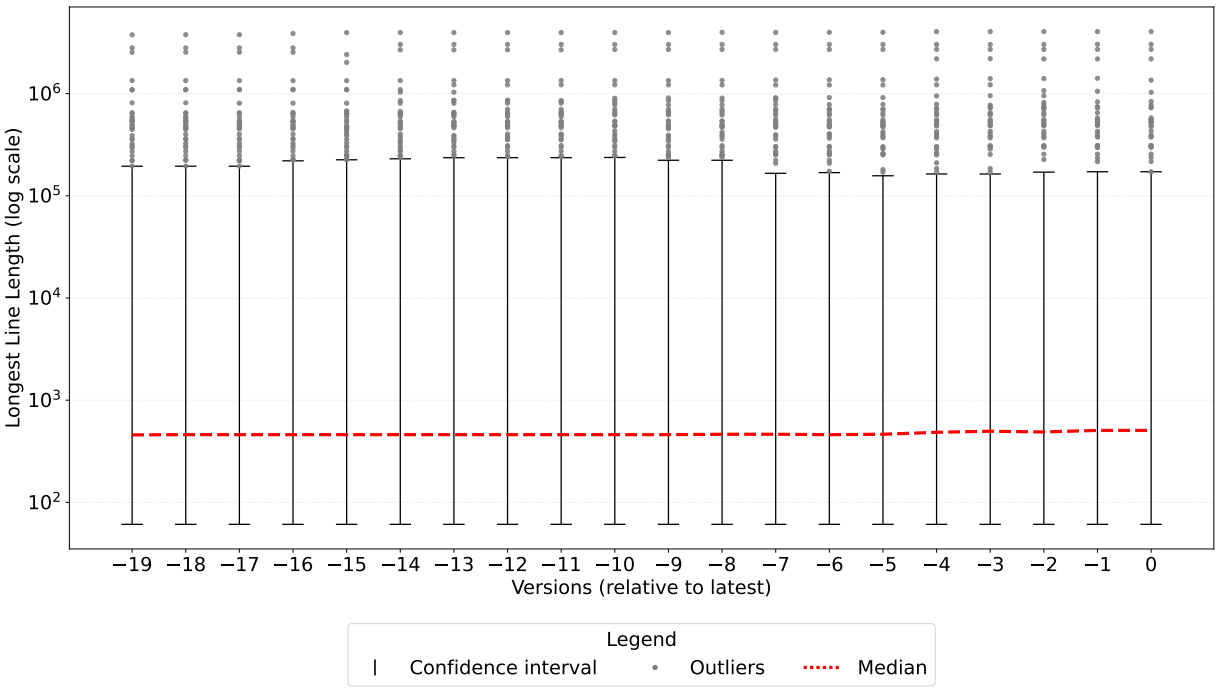

Figure 6.11: Longest Line Length Metric: Package Trend in Goodware Dataset.

The goodware dataset provides, for each of the 629 packages, the longest line length observed across all plain-text files in their 20 analyzed versions, as described in Section 5.1.1.

To visualize the distribution and evolutionary patterns of this metric across releases, we used Figures 6.11 and 6.12; where we can observe that the median remains stable at approximately 460 characters, indicating that the length of the longest line does not change significantly in the typical package. This stability is a useful property for anomaly detection, as a sudden large increase is precisely the signature left by the SSCAs considered.

An important caveat is that build artifacts, such as source map files (`.map`) and compiler-generated JavaScript, can legitimately contain very long lines, and are therefore a potential source of false positives for this metric. In this analysis, we chose to include the entire tarball contents without excluding any file type or directory a priori, because an attacker could inject malicious code anywhere within the tarball.

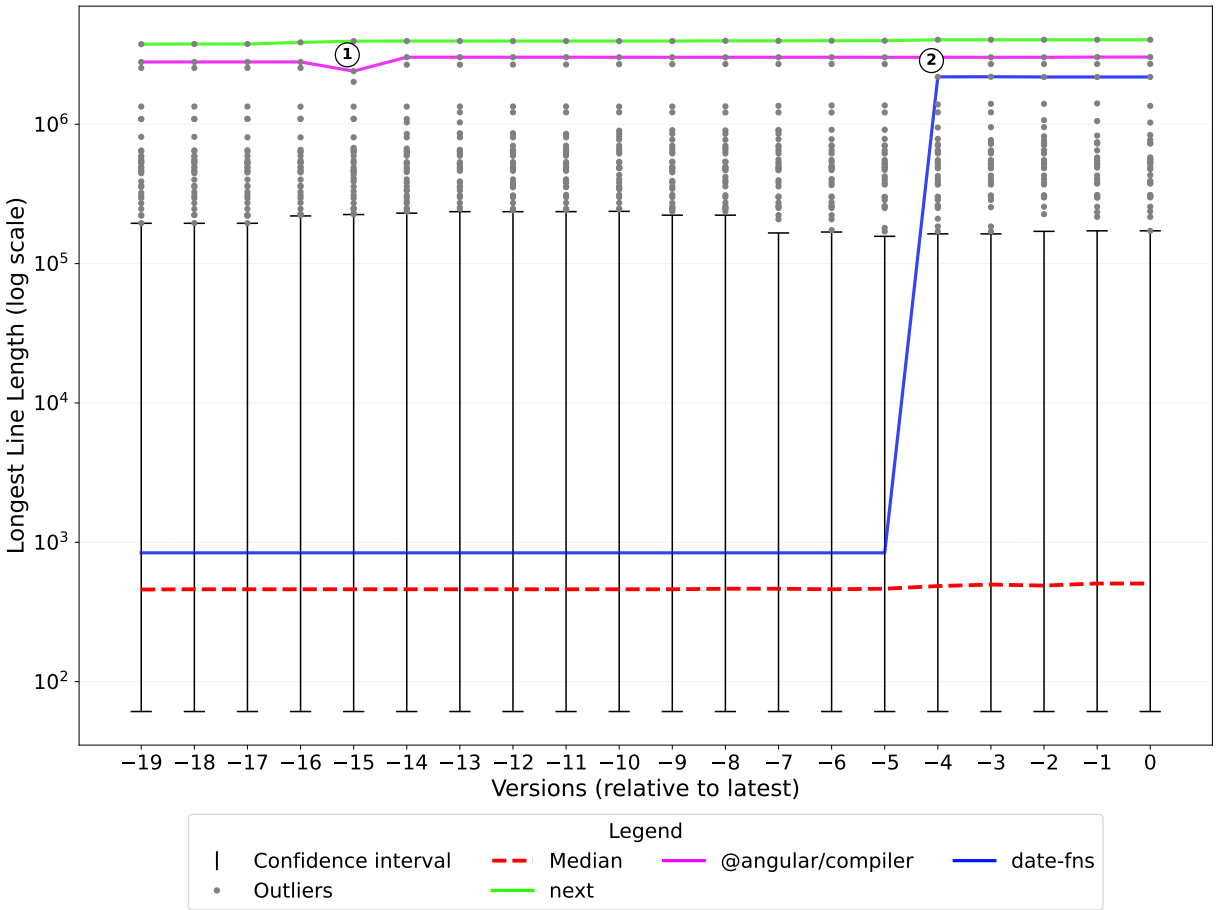

Figure 6.12: Longest Line Length Metric: Outliers Package Trend in Goodware Dataset.

As shown in Figure 6.12, from the goodware dataset we selected three packages for individual analysis as representative cases, as they exhibit outlier values directly attributable

to build artifacts.

next (solid green line) is a React full-stack framework [Ver]. It shows the most extreme outlier in the entire goodware dataset, with a longest line length of 4,041,545 characters, present in all 20 versions. This value originates from the file `app-page-turbo-experimental.runtime.dev.js.map`, a source map file consisting of a single line. The `next` package also contains other build artifacts with very long lines, such as `app-page.runtime.dev.js`, a generated JavaScript file containing 40 lines with the longest reaching 513,963 characters. Both files are located in the `dist` folder; however, it is not guaranteed that build artifacts are always present in any specific directory, as their location depends entirely on the individual developer choices.

@angular/compiler (solid pink line) is the official compiler of the Angular framework, responsible for tasks such as translating templates with Angular syntax [Teaa]. In all releases, the longest line originates from the file `compiler.mjs.map` in the `fesm2022` directory, consistently around 3,000,000 characters, making it the second most extreme outlier value in the goodware dataset after `next`. As indicated at ① in Figure 6.12, in version 21.0.0-rc.0 this value decreased to approximately 2,500,000 characters, reflecting a natural variation in the output of the build process for that release. This variability shows that even stable outlier values from build artifacts are not perfectly constant. `@angular/compiler` also contains other build artifacts, such as `compiler.mjs`, a generated JavaScript file containing approximately 30,000 lines with the longest reaching 14,825 characters.

date-fns (solid blue line) shows that build artifacts can be introduced in a new release without having been present in previous ones. Before version 3.6.0, this package was within the confidence intervals, but at ② in Figure 6.12 the metric increased to 2,188,784 characters, making it an outlier for all subsequent releases. This sudden increase was caused by the addition of `cdn.js.map` in the `locale` directory, consisting of 10 lines with the longest being 2,188,784 characters, making it the longest line found across all plain-text files in the package. It was introduced in version 3.6.0 together with approximately 400 new files for compatibility with older browsers, as already discussed in Section 6.3.1. The sudden change in this metric is superficially similar to what a malicious injection would produce, with the difference that this one is documented [tea26]. In this version, `date-fns` also includes other build artifacts, such as the file `cdn.js` containing 10 lines with the longest being 777,296 characters.

6.4.2 Comparison with the Malware Dataset

Attack A (Malicious Windows DLL, Section 4.2) includes a plain-text script `install.js`, which, however, does not contain the longest line in the compromised versions; and the malicious DLL was correctly excluded from the computation, as it is not a plain-text file.

Consequently, the longest line length of the compromised releases remained unchanged compared to the preceding legitimate versions, as shown in Figure 6.13, making this metric not indicative for this attack.

Attack B (Crypto attack, Section 4.3) adds a single malicious line of 76,438 characters to an existing `index.js`. As shown in Figure 6.14, this produces a notable increase in the metric, but not enough to position the compromised releases among the outlier values. The reason is that the goodware baseline includes packages with build artifacts containing lines of several tens of thousands in length, which sets the 95th-percentile bound at approximately 200,000 characters. The metric is therefore not indicative for this attack.

Attack C (`Shai-Hulud` 1.0, Section 4.4) introduces a malicious file named `bundle.js` containing a single line of 3,734,355 characters. As shown in Figure 6.15, this places the compromised versions among the outlier values of the goodware distribution. The metric is therefore effective for detecting Attack C.

Attack D (`Shai-Hulud` 2.0, Section 4.5) introduces a malicious file `bun_environment.js` containing a single line of 10,157,373 characters. As shown in Figure 6.16, the compromised releases not only enter the outlier range but far exceed the maximum outlier values in the goodware dataset. The metric is clearly effective for detecting Attack D.

In summary, despite this metric being influenced by the presence of build artifacts in legitimate packages, it effectively detects Attacks C and D, whose injected files contain a line long enough to exceed the 95th-percentile bound. Instead, it is not indicative for Attacks A and B, since the first scenario involves a binary file that is excluded from computation, while the malicious line of 76,438 characters introduced by Attack B remains within the upper confidence interval bound established in the goodware dataset.

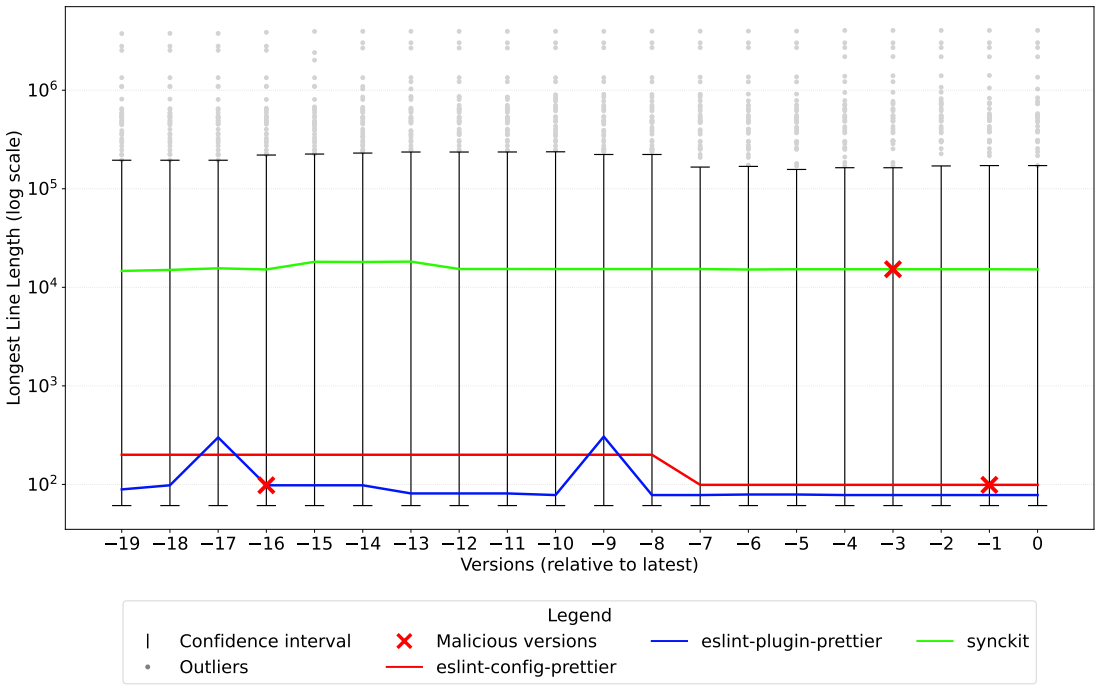

Figure 6.13: Longest Line Length Metric: Malware Comparison for Attack A.

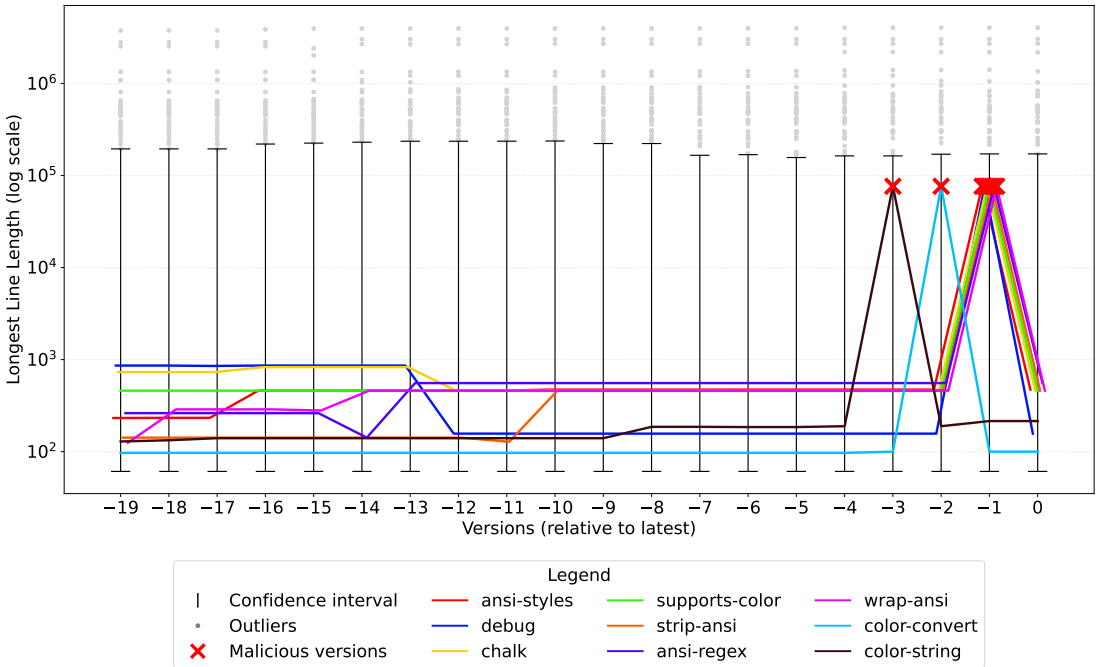

Figure 6.14: Longest Line Length Metric: Malware Comparison for Attack B.

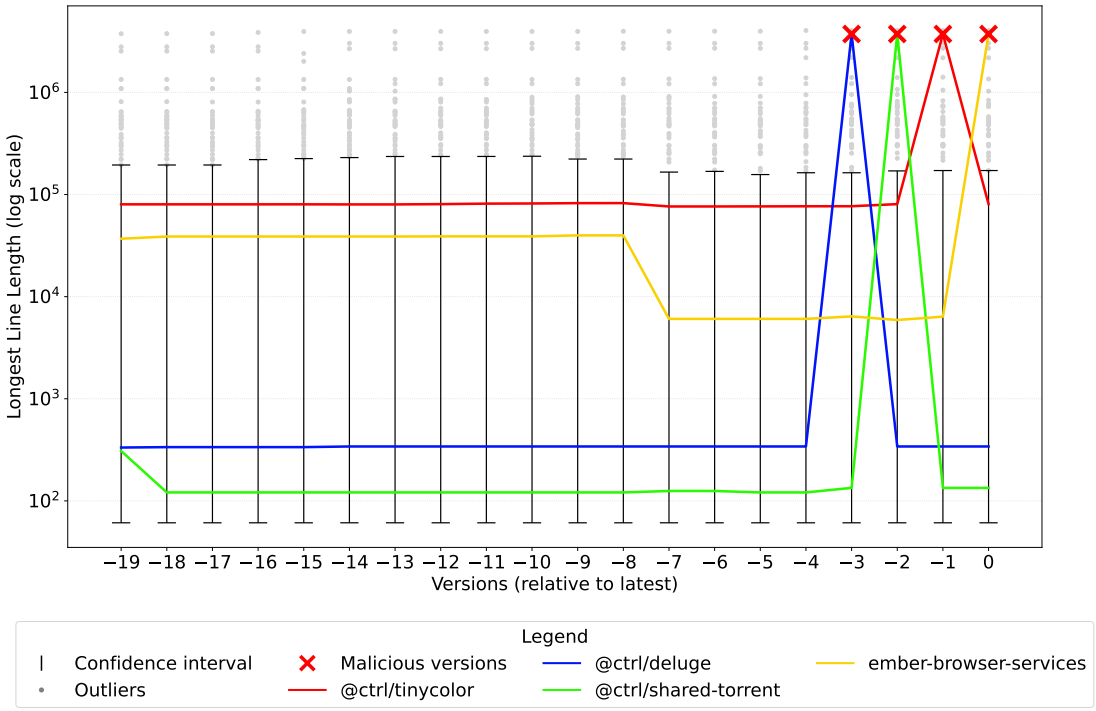

Figure 6.15: Longest Line Length Metric: Malware Comparison for Attack C.

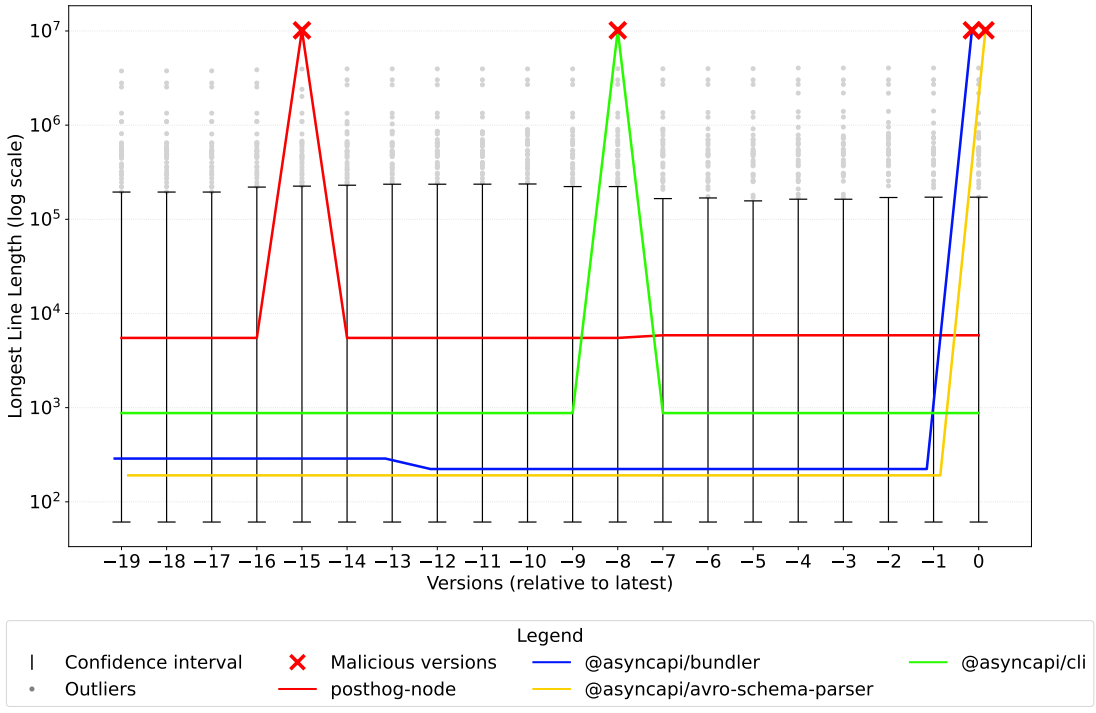

Figure 6.16: Longest Line Length Metric: Malware Comparison for Attack D.

6.5 Analysis Metric 4: Unique Hexadecimal Values

The fourth metric we analyzed is the count of unique hexadecimal values found across all plain-text files of an npm package version. It was motivated by the obfuscation technique used in Attacks B (Section 4.3) and D (Section 4.5), which encodes variable and function names as hexadecimal tokens to hide malicious behavior from static analysis, introducing obfuscated code that makes extensive use of such tokens. The intuition is that legitimate packages rarely encode their identifiers in this way, so a sudden increase in this count in a new version would constitute an anomalous signal worth investigating.

6.5.1 Analysis of the Goodware Dataset

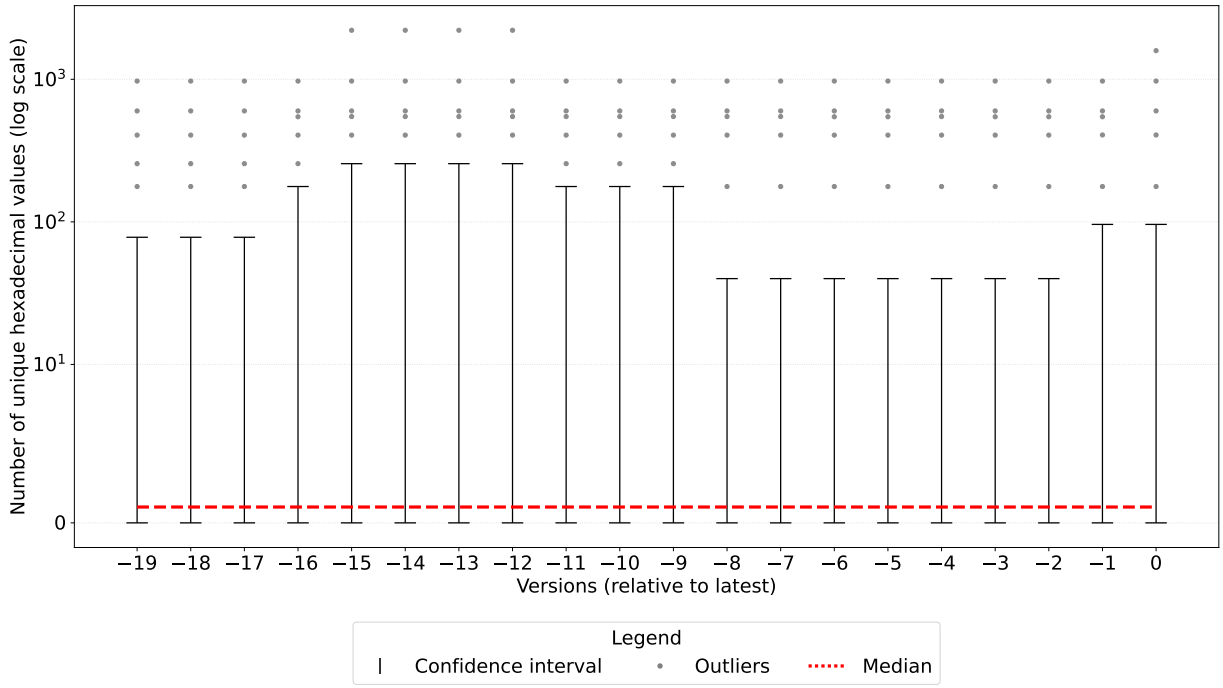

Figure 6.17: Unique Hexadecimal Values Metric: Package Trend in Goodware Dataset.

The goodware dataset provides, for each of the 629 packages, the count of unique hexadecimal values observed in their 20 analyzed versions, as described in Section 5.1.1. Of these, 520 have zero such values across all versions, and carry no information for this metric; the analysis in this section is therefore restricted to the remaining 109 packages (17.3%), each having at least one non-zero value in a given version. Within this subset, the median count

remains constant at 1 across all versions. To visualize the distribution and evolutionary patterns of this metric across releases, we used Figures 6.17 to 6.20. Among the considered packages, ten exhibit outlier or near-outlier values for at least one release and were selected for individual analysis. The three main causes of legitimate high hexadecimal counts are cryptographic constants (precomputed lookup tables and algorithm parameters), accidental inclusions, and build-tool-generated JavaScript files that use hexadecimal encoding for efficiency. Understanding these cases is essential to distinguish legitimate anomalies from malicious injections when evaluating the malware dataset. These ten packages are highlighted in the following figures with solid colored lines, categorizing them as follows based on their observed trends:

- **Increasing Trend** (Figure 6.18): illustrates highlighted packages with an increasing trend over time.
- **Decreasing Trend** (Figure 6.19): shows highlighted packages characterized by a reduction in this value.
- **Stable Trend** (Figure 6.20): displays highlighted packages maintaining a constant value throughout their history.

Figure 6.18 shows three packages that experienced a significant increase in this metric at a specific release: `crypto-js`, `polished`, and `xml2js`.

crypto-js (solid green line) is a cryptographic package that implements standard primitives, such as encryption algorithms and digital signatures [Mota]. In its first three versions from thirteen years ago (3.1.2 to 3.1.2-2), cryptographic constants were stored in decimal format, yielding a metric value of 0. As shown at ① in Figure 6.18, in version 3.1.2-3 the developers converted these values to hexadecimal, aligning with standard cryptographic specification practices, such as the Secure Hash Algorithm [GD95]. This brought the count to 546, an outlier maintained in subsequent versions. A further jump at ② corresponds to the introduction of the Blowfish algorithm in version 4.2.0, which added new hexadecimal constants and raised the count to 1,588, making it the most extreme outlier in the latest version.

polished (solid pink line) is a styling package for JavaScript [HS]. As seen at ③ in Figure 6.18, it jumped from 0 to 2,208 in version 4.0.0-beta.2, the most extreme outlier for four consecutive releases. This happened because the developers accidentally included a `.yarn` cache directory containing numerous files, including a build-tool-generated JavaScript file, as also discussed in Sections 6.2.1 and 6.3.1. When the folder was removed in version 4.0.4 [HS25], the count returned to 0. This is an example of an accidental anomaly traceable to a documented release note, distinguishable from a malicious injection because it is explained and subsequently corrected.

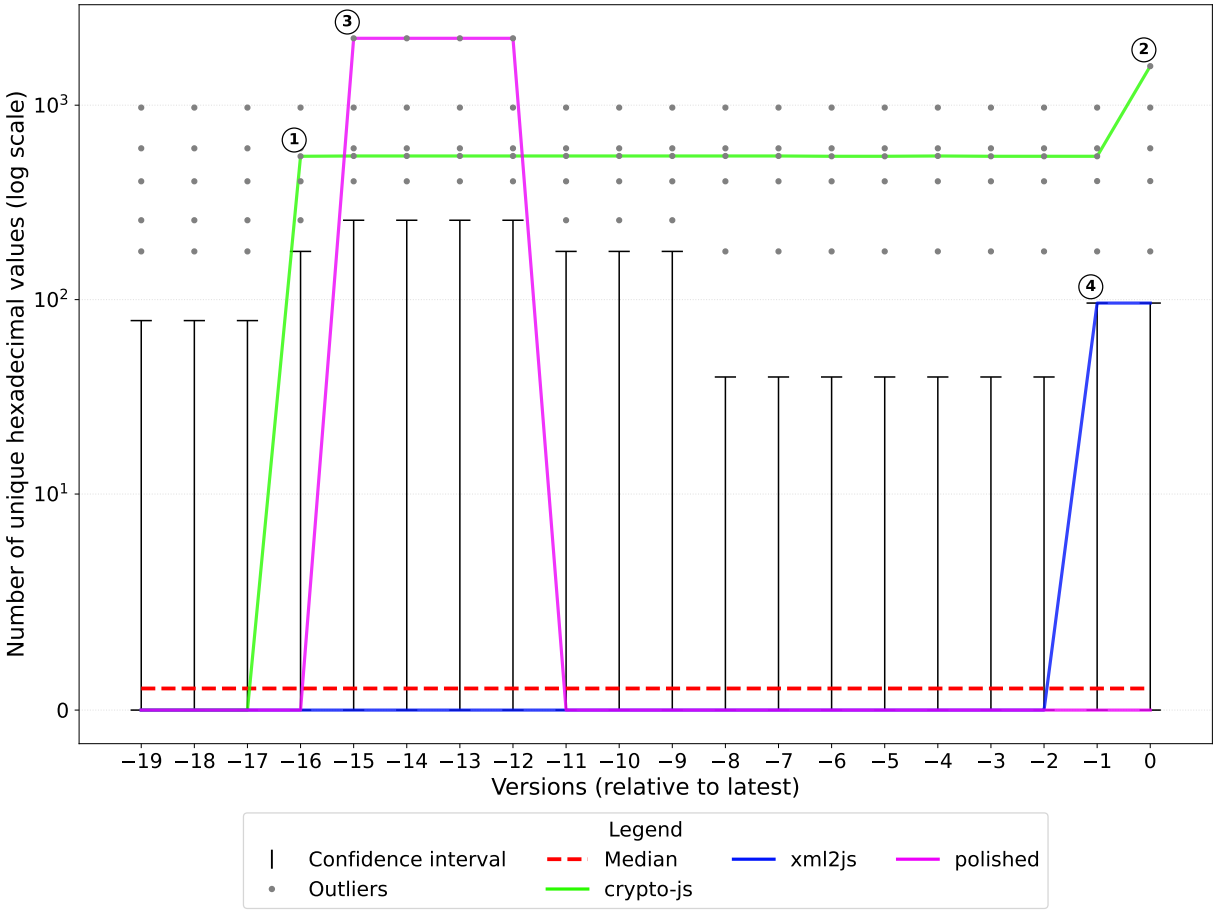

Figure 6.18: Unique Hexadecimal Values Metric: Outliers Package Increasing Trend in Goodware Dataset.

xml2js (solid blue line) went from 0 to 96 unique hexadecimal values at ④ in Figure 6.18. Starting from version 0.6.1, the developers included a JavaScript file automatically generated from OCaml source using the `js_of_ocaml` compiler, which produces output that uses such constants extensively for efficiency. The generated file remained in the package for the subsequent version 0.6.2.

Two packages experienced a notable decrease: `buffer-crc32` and `object-hash`, as shown in Figure 6.19.

buffer-crc32 (solid green line) implements the Cyclic Redundancy Check (CRC) algorithm for data integrity checking [Bre]. For efficiency reasons, the algorithm uses a CRC table, a matrix of 256 precalculated values that prevents the processor from performing complex mathematical calculations on every single byte. Until version 0.2.13, the CRC table values

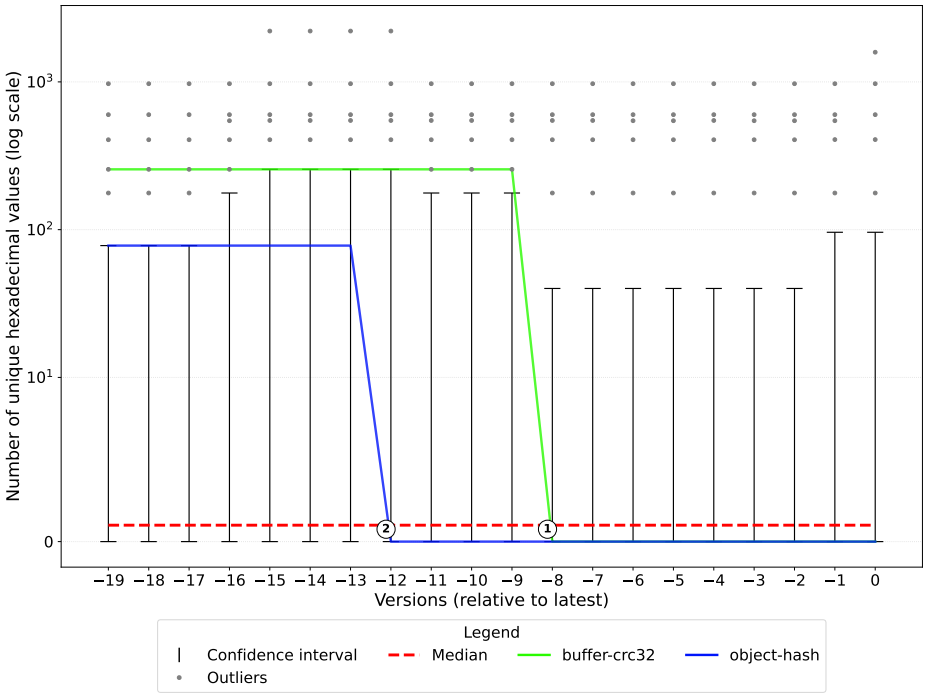

Figure 6.19: Unique Hexadecimal Values Metric: Outliers Package Decreasing Trend in Goodware Dataset.

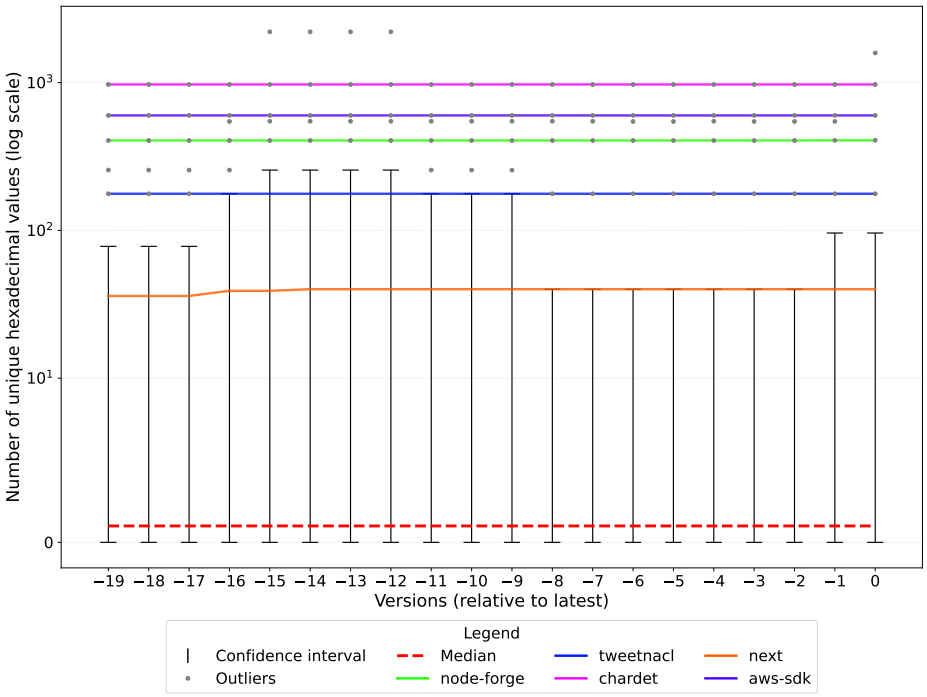

Figure 6.20: Unique Hexadecimal Values Metric: Outliers Package Stable Trend in Goodware Dataset.

were stored as hexadecimal constants. As shown at ① in Figure 6.19, from version 1.0.0-RC1 onward the developers switched to a build-tool-generated version of the table in which the values were converted to decimal for compactness. The count therefore dropped from 256 to 0 without any change in behavior.

object-hash (solid blue line) is a package that generates a hash from JavaScript objects [Hen]. From release -19 to -17 (1.1.0 to 1.1.2), it represented the upper limit of the confidence intervals, having 78 unique hexadecimal values, all located in the test file `object_hash_test.js`. At ② in Figure 6.19, starting from version 1.1.7, the test file was excluded from the published tarball for space optimization, as it is not essential for its functionality and not needed by end users. The count therefore dropped to 0.

Five packages maintain stable outlier or near-outlier values across all releases.

node-forge and **tweetnacl** are both cryptographic packages that use hexadecimal constants as part of their primitive implementations [Dig; Che], analogously to `crypto-js`. In Figure 6.20, `node-forge` (solid green line) and `tweetnacl` (solid blue line) exhibit 406 and 177 unique hexadecimal values, respectively, qualifying as outliers in either all or the vast majority of releases.

chardet (solid pink line) detects character encoding using byte maps stored as hexadecimal constants in `/lib/encoding/sbcs.js` [Yea], resulting in 972 values. As shown in Figure 6.20, this makes `chardet` the extreme outlier for most releases, surpassed only by `polished` during its four anomalous versions and by `crypto-js` in its most recent ones.

next (solid orange line) and **aws-sdk** (solid purple line) both contain hexadecimal values in generated JavaScript files bundled in their tarballs, in the same manner as `polished` and `buffer-crc32`. `next`, a React full-stack framework [Ver], has 40 such values consistently across all releases, representing the upper limit of the confidence intervals for versions -8 to -2, as shown in Figure 6.20. Instead `aws-sdk`, the official package for interacting with AWS cloud services [Ama], has 600 across all releases and is an outlier in every version.

6.5.2 Comparison with the Malware Dataset

Attack A (Malicious Windows DLL, Section 4.2) does not introduce any new hexadecimal values, as the malicious DLL is binary and excluded from computation, and the `install.js` script contains no hexadecimal tokens. As shown in Figure 6.21, the metric value is unchanged relative to the preceding legitimate versions, and the metric is not indicative for this attack.

Attack B (Crypto attack, Section 4.3) injects an obfuscated line of code into an existing `index.js` file. The resulting count of unique hexadecimal values reaches 455. As shown in Figure 6.22, this positions the malicious versions among the outliers of the goodware

distribution, making the metric effective for detecting Attack B.

Attack C (`Shai-Hulud` 1.0, Section 4.4) introduces a malicious file containing only one unique hexadecimal value. As shown in Figure 6.23, this marginal increase leaves the compromised releases within the confidence intervals of the goodware dataset, making the metric not indicative for this attack. This outcome is consistent with the nature of the SSCA as it does not rely on any obfuscation technique, but rather on the presence of clear-text malicious code.

Attack D (`Shai-Hulud` 2.0, Section 4.5) introduces an obfuscated file using the same hexadecimal encoding technique as Attack B. As shown in Figure 6.24, the count jumps to 122,535, far exceeding the maximum outlier values observed in the goodware dataset. The metric is clearly effective for detecting Attack D.

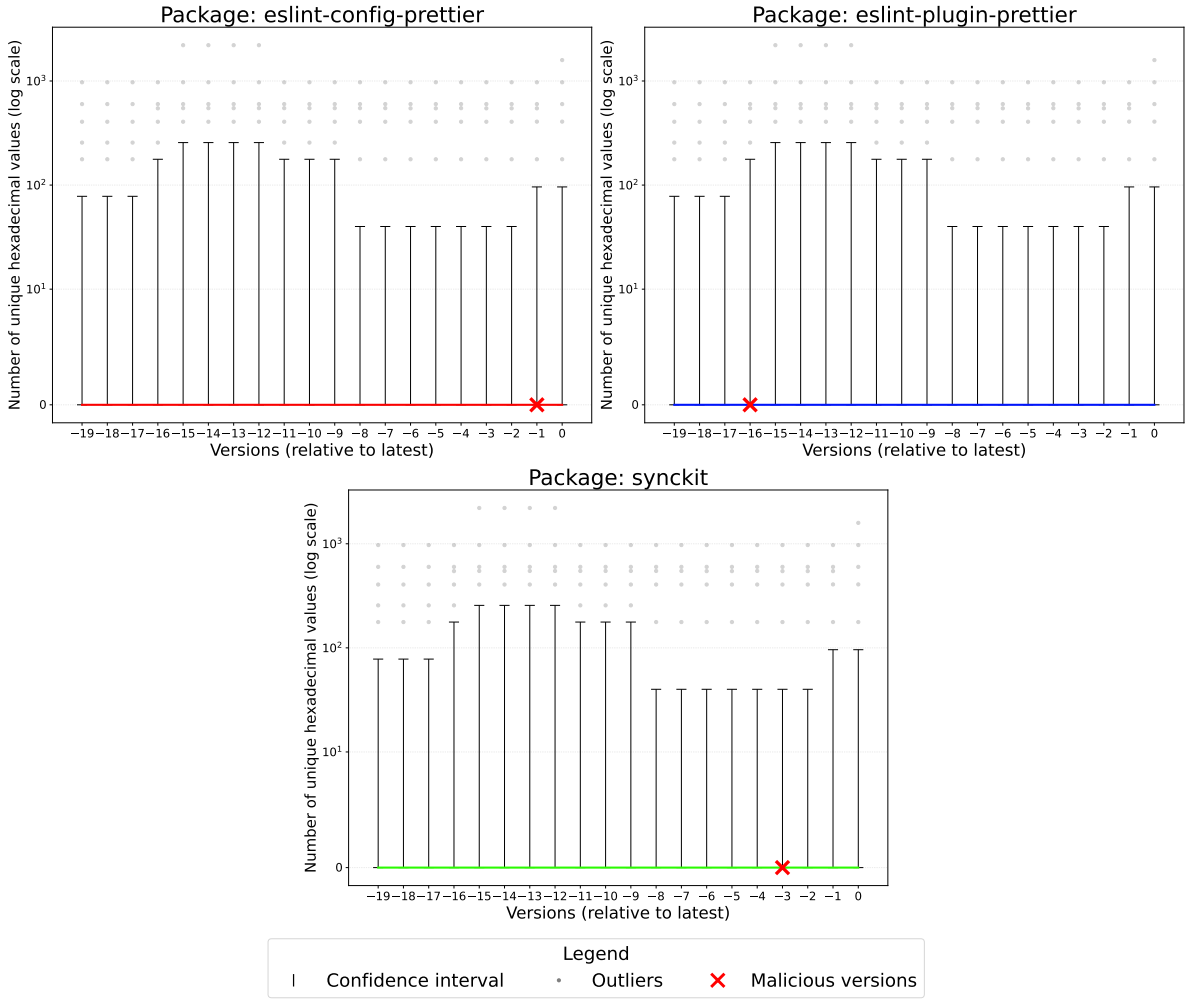

Figure 6.21: Unique Hexadecimal Values Metric: Malware Comparison for Attack A.

In summary, the *Unique Hexadecimal Values* metric is effective for attacks that use hexadecimal encoding as an obfuscation strategy (Attacks B and D), with detection strength scaling with the amount of encoded content, as Attack B produces 455 values, sufficient to cross the outlier boundary, while Attack D produces 122,535, an extreme anomaly with no parallel in the goodware baseline. It is not indicative for Attack A, whose malicious payload is binary, nor for Attack C, where the injected code is clear-text and does not rely on obfuscation.

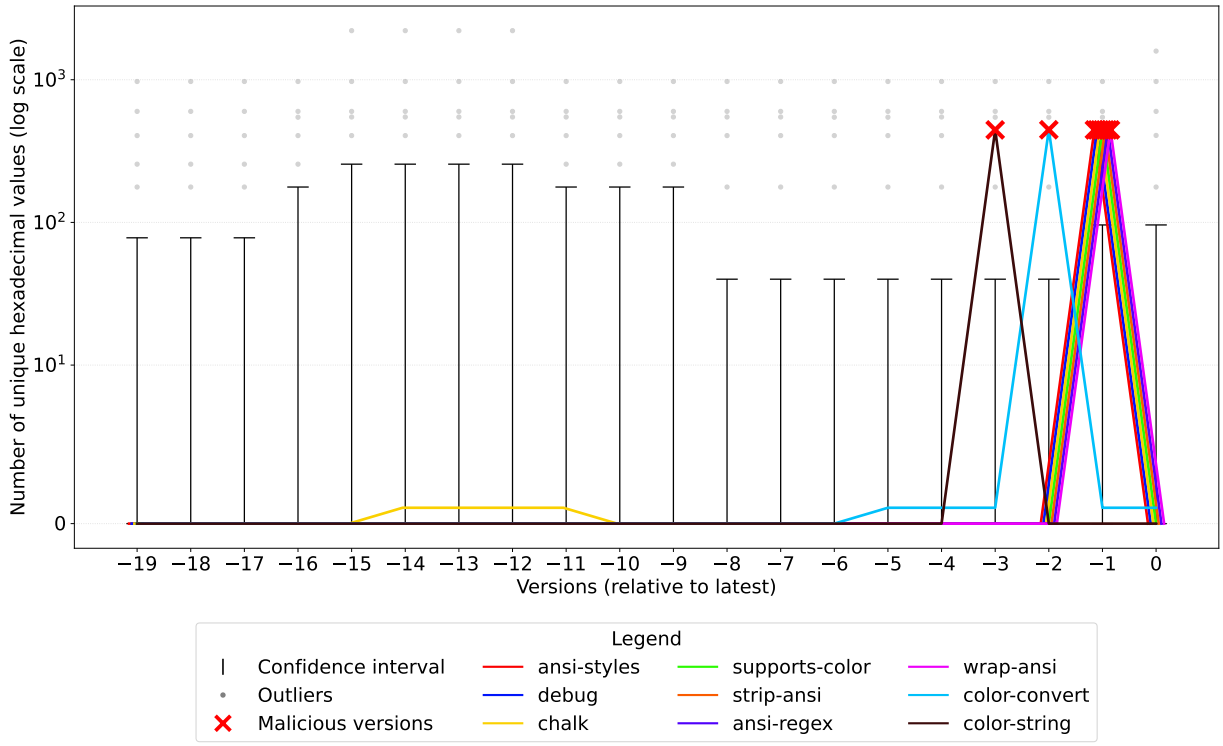

Figure 6.22: Unique Hexadecimal Values Metric: Malware Comparison for Attack B.

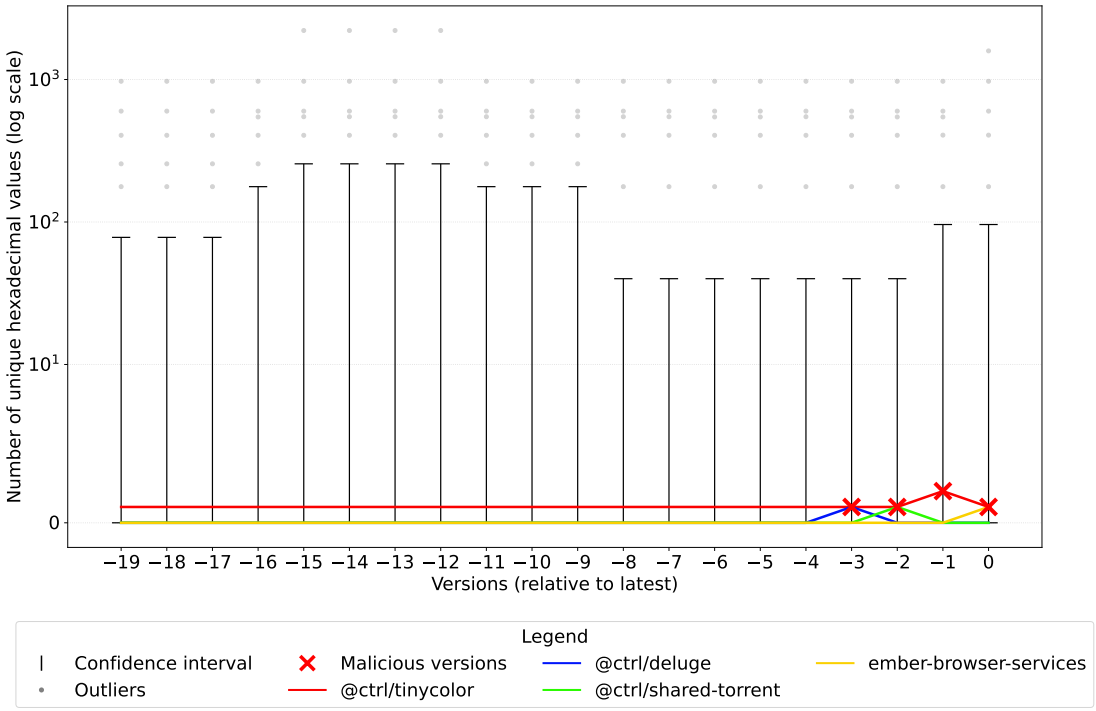

Figure 6.23: Unique Hexadecimal Values Metric: Malware Comparison for Attack C.

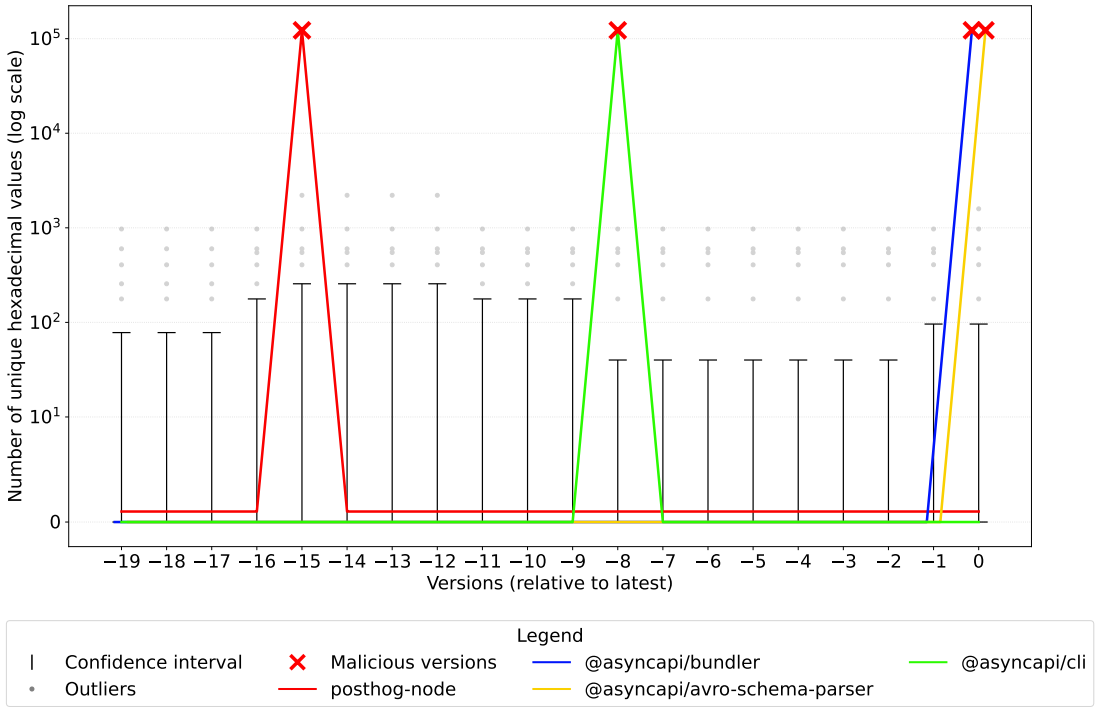

Figure 6.24: Unique Hexadecimal Values Metric: Malware Comparison for Attack D.

6.6 Analysis Metric 5: Unique Ethereum Addresses

The fifth and final metric is the count of unique Ethereum addresses found across all plain-text files of an npm package version. It was motivated by Attack B (Section 4.3), in which the malicious code embeds seven known smart contract Ethereum addresses to perform cryptocurrency transaction hijacking. The intuition is that Ethereum addresses are essentially absent from general-purpose npm packages, so even a small count would constitute an anomalous signal.

Each cryptocurrency defines its own address format, so this metric cannot be generalized across all currencies. We initially considered extending the analysis to Bitcoin addresses and their variants, generalizing this metric to *Unique Cryptocurrency Addresses*, but doing so produced a non-negligible number of false positives in the goodware dataset, as legitimate npm packages may contain strings that match the Bitcoin address format without being actual addresses. For instance, repeated digit sequences used as numerical constants or for testing purposes, such as `111111111122222222223333333333344`, are found in packages like `aws-sdk` across all their analyzed versions. Since Bitcoin addresses are not associated with any of the four SSCAs analyzed in this research, and no reliable library or regex rule was found capable of disambiguating true addresses from these coincidental matches, we chose to restrict the metric to Ethereum only, whose format is unambiguous, as shown in Section 6.6.1. This specificity is both a strength and a limitation, as it provides a reliable signal for attacks targeting Ethereum-based wallets, but offers no coverage for SSCAs that abuse other cryptocurrency ecosystems.

6.6.1 Analysis of the Goodware Dataset

For this metric, the goodware dataset provides the count of unique Ethereum addresses found in each of the 20 analyzed versions of every package, as described in Section 5.1.1. The distribution is so sparse that a trend figure is uninformative, and, across all 629 packages, only two contain at least one Ethereum address in a given version, namely `brace-expansion` and `dotenv-expand`, as also summarized in Table 5.3. Both cases are analyzed below to understand the reason for this presence.

brace-expansion provides shell-like brace expansion for JavaScript [Gru]. Starting from version 4.0.0 (the second-to-last available release), the tarball includes a file named `tea.yaml`, a configuration file required to register a project on the Tea protocol. The Tea protocol is an incentivization system for open-source projects that allows registered packages to earn TEA tokens proportional to their popularity and usage [Inc]. The `tea.yaml` file must include one or more Ethereum addresses to receive tokens and to define the identities of the project owners. This file is also present in the latest available version (4.0.1).

dotenv-expand extends the functionality of `dotenv` by allowing variable expansion in `.env` files [Motb]. Similarly to `brace-expansion`, starting from version 11.0.7 the tarball includes a `tea.yaml` file containing one Ethereum address. This file remained present for four consecutive releases, until version 12.0.3 (the most recent available), in which it was removed.

These two cases establish that the presence of even a single Ethereum address in an npm tarball is rare in the goodware dataset, and limited to a specific optional configuration file related to a third-party incentivization protocol, with no connection to the core functionality of the packages.

6.6.2 Comparison with the Malware Dataset

Given that only two packages in the goodware dataset contain one Ethereum address, the appearance of seven of them in the compromised versions of Attack B represents an important outlier. Because this is a value with no precedent in the goodware baseline, the metric is effective for detecting this specific attack.

Otherwise, the metric is not indicative for the remaining three attacks. For Attack A (Section 4.2), the malicious DLL is a binary file and is excluded from the computation, leaving the address count unchanged. For Attacks C and D (Sections 4.4 and 4.5), the malicious goal is credential and token exfiltration, not cryptocurrency theft; neither `bundle.js` nor `bun_environment.js` contains any Ethereum addresses, so the count remains at zero as in the legitimate versions.

This metric is therefore highly specific, as it detects the class of SSCA that target cryptocurrency wallets and provides no signal for other attack categories, making it a limited but reliable metric.

6.7 Wrap up Analysis Metrics

This section synthesizes the results of the analysis conducted on the five metrics defined in this chapter, evaluates their overall effectiveness in detecting the four attacks considered in this study, and discusses the proposed approach.

Table 6.1: Summary of Metric Effectiveness across the Four Attacks.

	Metric 1 File Types	Metric 2 Pkg Size	Metric 3 Line Len.	Metric 4 Hex Val.	Metric 5 Eth Addr.
Attack A - Malicious DLL	●	○	○	○	○
Attack B - Crypto Attack	○	○	○	●	●
Attack C - Shai-Hulud 1.0	○	●	●	○	○
Attack D - Shai-Hulud 2.0	○	●	●	●	○

Table 6.1 visually summarizes the detection capability of each metric for every SSCA. Filled green circles (●) and empty red ones (○) denote successful and unsuccessful detection, respectively.

From the analysis it can be observed that while every attack is detectable by at least one metric, none is capable of detecting all four SSCAs simultaneously. This reflects the diversity of the attacks, each with specific characteristics and leaving different traces that are not necessarily shared.

The detection logic for each metric can be summarized as follows.

As argued in Section 6.2, **Metric 1 - File Types** is effective when an attacker introduces a file type rarely found in npm packages, such as the PE binary in Attack A. Conversely, it is not indicative for SSCAs that introduce or modify extensions common to the ecosystem, such as JavaScript sources in the case of Attacks B, C, and D.

As discussed in Section 6.3, **Metric 2 - Package Size** is effective when the malicious files are large enough to push the total tarball size among the outliers of the goodware distribution. It succeeds for Attacks C (3.7 MB injection) and D (10 MB), but fails for Attacks A (1.2 MB, absorbed by the confidence interval) and B (a few kilobytes added to an existing file).

As seen in Section 6.4, **Metric 3 - Longest Line Length** is effective when the malicious code is written as a single extremely long line. It succeeds for Attacks C (3.7 million characters) and D (10 million), but not for Attack B (76,438), whose line length falls below the outlier threshold established also by the presence of build artifacts in legitimate packages. Furthermore, it fails for Attack A because its main malicious file is a binary and is therefore excluded from computation.

As detailed in Section 6.5, **Metric 4 - Unique Hexadecimal Values** is effective for attacks that introduce obfuscated code making extensive use of hexadecimal values to encode variable and function names, such as in the case of Attacks B (455 unique values) and D (122,535 values). Instead, it is not indicative for attacks that do not rely on this obfuscation technique, such as Attacks A and C.

As described in Section 6.6, **Metric 5 - Unique Ethereum Addresses** is highly specific to attacks that embed cryptocurrency wallet addresses for transaction hijacking. It effectively detects Attack B, which embeds seven known smart contract addresses, and provides no signal for the other three attacks, which have different goals and do not include any Ethereum address in their malicious versions.

Some limitations of the proposed approach should be noted.

First, the metrics were defined based on the specific characteristics of the four SSCAs analyzed. Diverse attacks with different properties not covered by this study may not be detectable through these metrics. Even an attacker aware of these metrics could craft a malicious payload specifically designed to evade all five, by injecting a small syntactically plausible file that avoids unusual extensions, long lines, hexadecimal identifiers, and Ethereum addresses. Therefore, it is necessary to extend this set of metrics to include additional, uncovered attack patterns.

Second, the goodware dataset covers 629 of the 1,000 most popular packages. Less popular ones may have different structural properties, and the confidence intervals derived from this dataset may not generalize perfectly to the entire npm ecosystem. Moreover, the goodware dataset is inherently heterogeneous, comprising packages that vary widely in size, purpose, and content. As a result, some legitimate packages show anomalous values across the considered metrics and must be analyzed individually and manually to understand their nature.

Finally, a monitoring system based on these metrics must maintain a history of 20 recent versions per package. For a registry the size of npm (over two million packages), this implies storing and updating a substantial amount of per-release data. However, the per-version computation is entirely local to the tarball and requires no external services or container environments, making the operational cost substantially lower than dynamic analysis or LLM-based approaches.

Chapter 7

Conclusion

This thesis investigated whether it is possible to detect anomalous releases in npm packages, using five lightweight, easily computable metrics extracted from tarballs, and tracking their evolution across 20 consecutive versions.

To this end, we analyzed in depth four SSCAs that affected the npm ecosystem in 2025, each sharing the same high-level attack model, namely the compromise of the account of a legitimate maintainer followed by the publication of a malicious update, but differing substantially in payload, technique, and goal. The Malicious Windows DLL attack (Attack A, Section 4.2) introduced a malicious Windows DLL to achieve remote code execution on victim machines, the Crypto Attack (Attack B, Section 4.3) embedded obfuscated code in a popular package to hijack cryptocurrency transactions in the browser, and `Shai-Hulud` 1.0 and 2.0 (Attacks C and D, Sections 4.4 and 4.5), the two variants of the `Shai-Hulud` worm, combined credential exfiltration and self-propagation across the registry. From the characteristics of these attacks we derived five metrics: *File Types*, *Package Size*, *Longest Line Length*, *Unique Hexadecimal Values*, and *Unique Ethereum Addresses*. These were evaluated against two datasets, a goodware dataset of 629 popular samples across their 20 most recent versions each, used to establish a baseline of normal evolutionary behavior, and a malware dataset of 20 packages compromised by the four SSCAs.

The results demonstrate that while each individual attack is detectable by at least one metric, no single one identifies all four simultaneously. The *File Types* metric proved effective only for Attack A, because the introduction of a Windows PE binary is essentially absent from the goodware baseline. The *Package Size* metric successfully detected Attacks C and D, both of which injected files of several megabytes, but was not sensitive enough for the smaller additions of Attacks A and B. The *Longest Line Length* metric detected the same two attacks, since both `Shai-Hulud` variants encoded their payloads as a single line of millions of characters; it is however inherently noisy due to build artifacts

in legitimate packages, which can themselves produce long lines. The *Unique Hexadecimal Values* metric proved effective for Attacks B and D, which both used hexadecimal-based identifier obfuscation to hide malicious logic from static analysis, producing counts of 455 and 122,535 unique values respectively, far above the goodware confidence intervals. The *Unique Ethereum Addresses* metric detected Attack B alone, whose goal of cryptocurrency theft required embedding known smart contract addresses in the malicious code in plain text; the remaining attacks, targeting credential exfiltration or remote code execution, left this metric unchanged.

Some limitations of this work should be taken into account when interpreting these results. The most important one is that the five metrics were derived from the specific characteristics of the four SSCAs analyzed. Diverse attacks with different properties not covered by this study may not be detectable through these metrics, as an attacker who is aware of such measures could craft a payload that evades all of them simultaneously. This limitation can be surpassed by extending the metric set to cover a broader range of attack patterns. The second limitation concerns the fact that the dataset covers 629 of the 1,000 most popular npm packages; less prominent ones may exhibit different properties, and the confidence intervals derived here may not transfer perfectly to the wider ecosystem. Moreover, the inherent heterogeneity of the dataset means that some legitimate packages present anomalous values for one or more metrics and require individual manual examination to understand their nature. Finally, a monitoring system based on these metrics must maintain a history of 20 recent releases for every package under observation. For a registry the size of npm, with over two million entries, this implies storing and continuously updating a substantial amount of per-release data. That said, the per-version computation is entirely local to the tarball and requires no sandboxed execution environments or external service calls, keeping the operational cost substantially lower than dynamic analysis or LLM-based approaches.

Despite these limitations, this work demonstrates that lightweight metrics, tracked across consecutive releases and evaluated against an empirically constructed baseline, are sufficient to detect all four real-world attacks studied. The two datasets and the two developed tools are publicly available to support reproducibility and future research. A natural direction for future work is to expand the metric set to cover attack patterns not represented in this study; promising candidates include behavioral indicators such as changes in the number of PMs, and releases occurring after unusually long periods of inactivity. Expanding the malware dataset to cover a broader range of SSCA families would also strengthen future evaluations and help assess whether the longitudinal approach generalizes to attack types beyond the four analyzed here.

Acknowledgements

I would like to thank my supervisors, Giovanni Lagorio and Luca Caviglione, for their guidance and assistance throughout the writing of this thesis, and all the professors I encountered during my university career for their advice and kindness. I am also deeply grateful to my family, Corrado, Laura and Francesco, my grandparents, and all my friends for their love and support.

Bibliography

[Aka24] Akamai. *XZ Utils Backdoor - Everything You Need to Know, and What You Can Do*. 2024. URL: https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know.

[Ama] Inc. Amazon Web Services. *Package aws-sdk*. URL: https://www.npmjs.com/package/aws-sdk.

[Bal19] A. Baldwin. *The package destroyer-of-worlds contained malicious code*. [Online]. May 2019. URL: https://www.npmjs.com/advisories/890.

[Ble22] BleepingComputer. *Dev corrupts NPM libs colors and faker breaking thousands of apps*. 2022. URL: https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps/.

[Bre] Brain J. Brennan. *Package buffer-crc32*. URL: https://www.npmjs.com/package/buffer-crc32.

[Bru] Max Brunsfeld. *Tree-sitter*. URL: https://tree-sitter.github.io/tree-sitter/.

[BC25] Moshe Siman Tov Bustan and Alexander Chailytko. *180+ npm Packages Hit in Major Supply Chain Attack*. 2025. URL: https://www.ox.security/blog/npm-2-0-hack-40-npm-packages-hit-in-major-supply-chain-attack/.

[BC14] Vitalik Buterin and Ethereum Contributors. *Ethereum Documentation*. 2014. URL: https://ethereum.org/learn/.

[Che] Dmitry Chestnykh. *Package tweetnacl*. URL: https://www.npmjs.com/package/tweetnacl.

[CC] Matteo Collina and David Mark Clements. *Package pino*. URL: https://www.npmjs.com/package/pino.

[Cona] Mozilla Contributors. *Lexical grammar JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar.

[con] Node.js contributors. *Package node-addon-api*. URL: https://www.npmjs.com/package/node-addon-api.

[Conb] Webpack Contributors. *Webpack*. URL: https://webpack.js.org/concepts/.

[Cut25] Gianpietro Cutolo. *Shai-Hulud 2.0: Aggressive, Automated, and Fast Spreading*. 2025. URL: https://www.netskope.com/blog/shai-hulud-2-0-aggressive-automated-one-of-fastest-spreading-npm-supply-chain-attacks-ever-observed.

[Dat23] Datadog. Mar. 2023. URL: https://github.com/datadog/malicious-software-packages-dataset.

[Def] DefinitelyTyped. *Package @types/node*. URL: https://www.npmjs.com/package/@types/node.

[Dig] Inc. Digital Bazaar. *Package node-forge*. URL: https://www.npmjs.com/package/node-forge.

[Dua+20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. *Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages*. 2020. arXiv: 2002.01139 [cs.CR]. URL: https://arxiv.org/abs/2002.01139.

[Eer] Matthew Eernisse. *Package Jake*. URL: https://www.npmjs.com/package/jake.

[Fou] Python Software Foundation. *Python re Library*. URL: https://docs.python.org/3/library/re.html.

[Ful] Lovell Fuller. *Package sharp*. URL: https://www.npmjs.com/package/sharp.

[GD95] Patrick Gallagher and Acting Director. "Secure hash standard (shs)". In: *Fips pub* 180 (1995), p. 183.

[Gil26] Patrick Gillespie. *Figlet Release Notes*. 2026. URL: https://github.com/patorjk/figlet.js/releases.

[Gil] Patrick Gillespie. *Package figlet*. URL: https://www.npmjs.com/package/figlet.

[GAH23] Betul Gokkaya, Leonardo Aniello, and Basel Halak. *Software supply chain: review of attacks, risk assessment strategies and security controls*. May 2023. DOI: 10.48550/arXiv.2305.14157.

[Goo] Google. *Magika*. URL: https://github.com/google/magika.

[Gos+20] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. "Investigating The Reproducibility of npm Packages". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 677–681. DOI: 10.1109/ICSME46990.2020.00071.

[Gru] Julian Gruber. *Package brace-expansion*. npm Package Registry. URL: https://www.npmjs.com/package/brace-expansion.

[Guo+23] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. *An Empirical Study of Malicious Code In PyPI Ecosystem*. 2023. arXiv: 2309.11021 [cs.SE]. URL: https://arxiv.org/abs/2309.11021.

[HH25] Asaf Henig and Cameron Hyde. *Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk*. 2025. URL: https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/.

[Hen] Basim Hennawi. *Package object-hash*. URL: https://www.npmjs.com/package/object-hash.

[HS25] Brian Hough and Max Stoiber. *Polished Release Notes*. 2025. URL: https://github.com/styled-components/polished/releases?page=2.

[HS] Brian Hough and Max Stoiber. *Package polished*. URL: https://www.npmjs.com/package/polished.

[Hua+24a] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. *DONAPI: Malicious npm Packages Detector using Behavior Sequence Knowledge Mapping*. 2024. arXiv: 2403.08334 [cs.CR]. URL: https://arxiv.org/abs/2403.08334.

[Hua+24b] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. "SpiderScan: Practical Detection of Malicious npm Packages Based on Graph-Based Behavior Modeling and Matching". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1146–1158. ISBN: 9798400712487. DOI: 10.1145/3691620.3695492. URL: https://doi.org/10.1145/3691620.3695492.

[Inc] Tea Inc. *Tea Protocol Documentation*. URL: https://docs.tea.xyz/tea.

[Lab25] Endor Labs. *CVE-2025-54313: eslint-config-prettier Compromise — High Severity but Windows-Only*. 2025. URL: https://www.endorlabs.com/learn/cve-2025-54313-eslint-config-prettier-compromise----high-severity-but-windows-only.

[Mai] Cheerio Maintainers. *Package cheerio*. URL: https://github.com/cheeriojs/cheerio/releases?page=2.

[Mat25] Nicolas Matthee. *The npm Supply Chain Attack of September 2025*. 2025. URL: https://www.prventi.com/blog/the-npm-supply-chain-attack-of-september-2025.

[MDN] MDN. *Minification*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Minification.

[Mey25] Darren Meyer. *PyPI Supply Chain Attack Uncovered: Colorama and Colorizr Name Confusion*. 2025. URL: https://checkmarx.com/zero-post/python-pypi-supply-chain-attack-colorama/.

[Mic] Microsoft. *Package typescript*. URL: https://www.npmjs.com/package/typescript.

[Moo+21] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. "Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, pp. 569–580. DOI: 10.1109/DSN48987.2021.00065.

[Mota] Jeff Mott. *Package crypto-js*. URL: https://www.npmjs.com/package/crypto-js.

[Motb] Scott Motte. *Package dotenv-expand*. npm Package Registry. URL: https://www.npmjs.com/package/dotenv-expand.

[npm26] Inc. npm. *npm Documentation*. 2026. URL: https://docs.npmjs.com/.

[Ohm+20] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. *Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks*. 2020. arXiv: 2005.09535 [cs.CR]. URL: https://arxiv.org/abs/2005.09535.

[Pat+25] Jinish Patel, Joseph Reiner, Brenden Stilwell, Abdullah Wahbeh, and Raed Seetan. "Leveraging K-Means Clustering and Z-Score for Anomaly Detection in Bitcoin Transactions". In: *Informatics* 12 (Apr. 2025), p. 43. DOI: 10.3390/informatics12020043.

[Pin+23] Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. *A Large Scale Analysis of Semantic Versioning in npm*. 2023. arXiv: 2304.00394 [cs.SE]. URL: https://arxiv.org/abs/2304.00394.

[Pre] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: https://semver.org/.

[Sec] Truffle Security. *TruffleHog*. URL: https://trufflesecurity.com/trufflehog.

[Sha] Remy Sharp. *Package nodemon*. URL: https://www.npmjs.com/package/nodemon.

[Sny21] Snyk. *Typosquatting Attack*. 2021. URL: https://snyk.io/blog/typosquatting-attacks/.

[Sum26] Jarred Sumner. *Bun*. 2026. URL: https://bun.com/.

[Teaa] Angular Team. *Package @angular/compiler*. URL: https://www.npmjs.com/package/@angular/compiler.

[tea26] Date-fns team. *Date-fns Release Notes*. 2026. URL: https://github.com/date-fns/date-fns/releases.

[tea] Date-fns team. *Package date-fns*. URL: https://www.npmjs.com/package/date-fns.

[Tea25a] Editorial Team. *npm registry attacked by secret-stealing worm*. 2025. URL: https://www.kaspersky.com/blog/tinycolor-shai-hulud-supply-chain-attack/54315/.

[Tea25b] HelixGuard Team. *Shai-Hulud Returns: Over 1K npm Packages and 27K+ Github Repos infected via Fake Bun Runtime Within Hours*. 2025. URL: https://helixguard.ai/blog/malicious-sha1hulud-2025-11-24/.

[Teab] Matplotlib Development Team. *Matplotlib Documentation*. URL: https://matplotlib.org/.

[Teac] Pandas Development Team. *Pandas Documentation*. URL: https://pandas.pydata.org.

[Tec18] TechTarget. *Compromised NPM package highlights open source trouble*. 2018. URL: https://www.techtarget.com/searchsecurity/news/252453398/Compromised-NPM-package-highlights-open-source-trouble.

[Tob] Christer Tobjörk. *Package node-releases*. URL: https://www.npmjs.com/package/node-releases.

[Tri26] F.-R. Tristan. *List of top 10000 npm packages*. 2026. URL: https://tristan-f-r.github.io/npm-rank/PACKAGES.html.

[Typ] Typicode. *Package husky*. URL: https://www.npmjs.com/package/husky.

[VB] Rod Vagg and Benjamin Byholm. *Package nan*. URL: https://www.npmjs.com/package/nan.

[Ver] Vercel. *Package next*. URL: https://www.npmjs.com/package/next.

[Wat] Shinnosuke Watanabe. *Package spdx-license-ids*. URL: https://www.npmjs.com/package/spdx-license-ids.

[Wei+25] Felix Weissberg, Lukas Pirch, Erik Imgrund, Jonas Möller, Thorsten Eisenhofer, and Konrad Rieck. "LLM-based Vulnerability Discovery through the Lens of Code Metrics". In: *arXiv preprint arXiv:2509.19117* (2025).

[Yea] Graeme Yeates. *Package chardet*. URL: https://www.npmjs.com/package/chardet.

[Yee] Jecelyn Yeen. *What are source maps?* URL: https://web.dev/articles/source-maps.

[Zah+25] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. *Leveraging Large Language Models to Detect npm Malicious Packages*. 2025. arXiv: 2403.12196 [cs.CR]. URL: https://arxiv.org/abs/2403.12196.

[Zor25] Marco Zoratti. "Valutazione di Metriche per il Rilevamento di Attacchi al Codice Sorgente". In: (2025). URL: https://unire.unige.it/handle/123456789/12801.