



**Technische Hochschule
Brandenburg**
University of
Applied Sciences

Technische Hochschule Brandenburg

Faculty of Computer Science

Fingerprinting-Merkmale für Bilddaten unter Einfluss von Information Hiding

Submitted in partial fulfillment of the requirements for the degree of
Master of Science (M.Sc.)

by

Fabian Loewe

Matrikelnummer: 20202415

First Examiner

Prof. Dr. Claus Vielhauer

Second Examiner

Benedict Michaelis, M.Sc.

Abstract

In dieser Masterarbeit wird zunächst ein Überblick zum Information Hiding mithilfe von Bilddateien verschafft. Der Fokus wird auf die Verwendung der beschriebenen Techniken im öffentlichen Raum durch Einbettungswerkzeuge und Malware gesetzt. Eine Untersuchung aktueller Malware-Vorkommnisse zeigt die Relevanz der Thematik und verschafft eine kategorisierte Übersicht für weitere Untersuchungen. Anhand der Funde werden Einbettungswerkzeug, die algorithmisch verwandte Verfahren zu den untersuchten Malwares anwenden, und Forschungsdaten mit Cover- und Stego-Bild-Paaren ausgewählt. Die Bildpaare werden daraufhin mit einem Werkzeug verglichen und aus den Ergebnissen steganografische Merkmale extrahiert, welche Hinweise auf die Anwendung bestimmter Algorithmen, Bildverarbeitungsbibliotheken und weitere Informationen liefern. Schließlich werden im Ausblick weitere Forschungsrichtungen im Bereich des Information Hiding vorgestellt.

Declaration

Suppose you are writing a thesis; you probably need this bit to confirm that you wrote it all by yourself. This template adds the `signature-required` CSS class, which adds a nice line where you can write your name.

If you are not writing a thesis, just delete this whole section.

Potsdam, 1.1.2024_

Fabian Loewe

Table of Contents

Abstract	2
1. Einleitung	7
1.1. Motivation	7
1.2. Zielsetzung	8
1.3. Forschungsfragen und Hypthoesen	8
1.4. Aufbau der Arbeit	9
2. Theoretische Grundlagen	10
2.1. Information Hiding	10
2.1.1. Anwendungsgebiete	10
2.1.2. Kriterien	10
2.1.3. Verfahren	12
2.2. Merkmalsanalyse	13
3. Methodik	15
3.1. Suchmaschinen	15
3.2. Literaturrecherche zu Malware-Vorkommnissen im Zusammenhang mit Bild- Steganografie	15
3.2.1. Manuelle Recherche	16
3.2.2. Automatisierte Sammlung aus Datenbanken	16
3.3. Literaturrecherche zu Steganografie- und Wasserzeichen-Tools	27
3.3.1. Steganografie-Software	27
3.3.2. Wasserzeichen-Software	28
3.4. Literaturrecherche zu Forschungsdaten für Bild-Steganografie	28
3.5. Implementierung eines Prototyps zur Merkmalsfindung steganografischer Bilddaten	29
3.5.1. Merkmalskategorien	29
3.5.2. Analyse-Tools	29
3.5.3. Vorstellung der Analysen	30
3.5.4. Implementierung eines prototypischen Detektors	35
4. Ergebnisse	38
4.1. Malware-Vorkommnisse im Zusammenhang mit Steganografie	38
4.2. Steganografie- und Wasserzeichen-Software	44
4.2.1. Übersicht zu Steganografie-Software	44
4.2.2. Übersicht zu Wasserzeichen-Software	45
4.2.3. Gesamtübersicht	46
4.3. Verwendeter Bild-Datensatz	46
4.4. Übersicht von Merkmalen steganografischer Bilddaten	47
4.5. Auswertung des Detektors	47
5. Zusammenfassung	48
6. Ausblick	49

Bibliografie	50
Anhang A: Tabellen	53
Anhang B: Abbildungen	54
Anhang C: Quellcode	55
Anhang D: Sonstiges	56

Chapter 1. Einleitung

Informationen versteckt in Schrift, Bild oder, allgemeiner, einem anderen Objekt zu übermitteln ist eine Jahrhunderte alte Technik, welche als Steganografie bereits seit 2500 Jahren bekannt ist. Der Begriff *Steganografie* leitet sich vom griechischen *steganós* (στεγανός) und *-graphia* (γραφία) ab, was grob übersetzt für *verborgen* und *Schrift* steht. [1]

So nutzte beispielsweise Griechen im Krieg mit Persien im fünften Jahrhundert versteckte Nachrichten, um nicht von den Persern eingenommen zu werden. Die schottische Königin Mary griff im 16. Jahrhundert auf Kryptografie und Steganografie zurück, um ihre Briefe vor ungewünschten Lesern zu schützen. Im Jahr 1499 veröffentlichte der deutsche Abt Johannes Trithemius sein Werk *Steganographia*, welches erstmals, soweit bekannt, Methoden zum Verstecken von Nachrichten in Schrift beschäftigt. [2]

Insbesondere mit der Entwicklung von Computern nahm Steganografie als Forschungsbereich final Einzug in die Wissenschaft, wobei großes Interesse besonders bei der Einbettung von Informationen in Medien wie Bild- und Audiodaten aufgekommen. Die Grundlagen der noch heute relevanten Methoden wurden bereits in den 1990er bis 2000er Jahren entwickelt und werden in den kommenden Kapiteln dieser Arbeit genauer vorgestellt.

1.1. Motivation

Für Entwickler von Schadsoftware, auch bekannt als Malware, ist stets wichtig, im Wettstreit mit den Gesetzeshütern einen Schritt voraus zu sein. Dazu wird inzwischen die Technik der Steganografie immer öfter aufgegriffen, um Daten wie weiteren Schadcode für Multi-Layer- oder Befehle für sogenannte Command-and-Control-Malware (kurz C&C) in das von der Schadsoftware befallene System einzuschleusen oder gestohlene Daten wie Passwörter aus dem System auszuschleusen, ohne dabei bei IT-Sicherheitssystem wie Firewalls oder Antivirenprogrammen aufzufallen. Die genannte Vorgänge sind jeweils als Dateninfiltration und Datenexfiltration bekannt. [3, 4]

Die Verwendung von Steganografie in Bilddateien kann bereits in vielen Fällen erkannt werden. Jedoch folgt darauf meist keine genauere Zuordnung zu einem bestimmten Algorithmus, einer möglicherweise verwendeten Bibliothek zur Verarbeitung von Bilddaten oder sogar einer bestimmten Malware oder Malware-Familie. Eine derartige Zuordnung kann vielfältige Informationen zum Angriffshergang, Beschaffenheit und Ursprung der Malware wie auch zur Herkunft der damit operierenden Täter liefern, die für die strafrechtliche Verfolgung von großer Bedeutung sein können.

Neben der Verwendung durch Malware wird Steganografie zudem auch für das Einbetten sogenannter digitaler Wasserzeichen genutzt, bei dem Bilddateien mit einem sichtbaren oder versteckten Code versehen werden, um ungewünschte Distributionen oder Manipulationen des Bildes nachverfolgen zu können. Dies ist wichtig, um Urheberrechte zu schützen und deren Bruch durch beispielsweise Piraterie ahnden zu können. Die Analyse von potenziell mit Wasserzeichen gekennzeichneten Bildern kann also wie bei Malware Aufschluss auf Herkunft und konkrete Art der Einbettung liefern, um beispielsweise Robustheit des verwendeten Algorithmus zu testen und sicherzustellen, dass bei der Einbettung keine privaten Informationen preisgegeben werden. [5]

1.2. Zielsetzung

Die Zielstellung dieser Arbeit konzentriert sich auf die folgenden Punkte:

1. Erstellung einer umfänglichen Liste von Malware-Vorkommnissen, in denen Steganografie mit Bildern als Trägermedium verwendet wurde
2. Erstellung einer Auswahlliste an Steganografie-Software, die vergleichbar in ihrer Funktionalität zu den gefundenen Malwares sind, sowie einer Auswahlliste an Wasserzeichen-Software
3. Recherche und/oder Aufbau von Forschungsdaten aus Cover- und Stego-Bilder-Paaren
4. Entwicklung eines Prototyps eines Werkzeugs zur Identifikation von Merkmalen für den Einsatz bestimmter Steganografieverfahren, Bildverarbeitungsbibliotheken und weitere

Durch das Erreichen der genannten Ziele soll ein erster Ansatz zur Identifikation von Steganografie-Tools basierend auf deren verwendeter Bilder erarbeitet werden, welcher sowohl für den Einsatz mit Malware als auch mit Wasserzeichen weiterausgebaut werden kann, wobei dies nicht mehr im Fokus dieser Arbeit liegt.

Da die Menge der Malware-Vorkommnisse erwartbar weit größer als die Menge der Steganografie- und Wasserzeichen-Tools sein dürfte, werden erstere nur mittels Literaturrecherche, während letztere auch praktisch durch Tests ausgewertet werden.

Die beschriebene Ziele werden fortan im Rahmen dieser Arbeit als **Z.1** bis **Z.4** referenziert.

1.3. Forschungsfragen und Hypthoesen

Aus der Zielsetzung lassen sich folgende Forschungsfragen und zugehörigen Hypthosen ableiten:

Frage: Können Werkzeuge zur Einbettung von Daten mittels Steganografie nur anhand ihrer Ausgabe-Bilder und der Original-Bilder automatisiert identifiziert werden?

Hypothese: Ja, es lassen sich immer wiederkehrende Merkmale in den Bildern nachweisen, die mit dem verwendeten Einbettungswerkzeug in Zusammenhang stehen.

Frage: Gibt es eine Schnittmenge bzgl. der Merkmale der verwendeten Steganografie-Algorithmen und technischen Umsetzung zwischen Malwares, Steganografie- und Wasserzeichen-Tools?

Hypothese: Ja, es kann eine Schnittmenge zumindest in der Kategorie der Steganografie-Algorithmen und der zur Umsetzung verwendeten Programmiersprachen und Bibliotheken abgesteckt werden.

Frage: Worin unterscheiden sich die drei Kategorien bzgl. der Merkmale?

Hypothese: Die Malwares legen einen größeren Wert auf Performance als auf Robustheit, während Wassezeichen-Tools umgekehrt gewichtet sind. Steganografie-Tools sind je nach Gewichtung der Autoren aufgestellt.

Die beschriebene Fragen werden fortan im Rahmen dieser Arbeit als **F.1** bis **F.3** referenziert.

1.4. Aufbau der Arbeit

Im folgenden Kapitel [Theoretische Grundlagen](#) werden die theoretischen Grundlagen kurz dargelegt, ohne tiefergehend auf die mathematische Basis einzugehen, da im Rahmen dieser Arbeit keine eigenen steganografischen Algorithmen entwickelt werden. Im Kapitel [Methodik](#) werden das Vorgehen zu den Literaturrecherchen und der Entwicklung von Programmen zur Erreichung von **Z.5** dokumentiert. Das Kapitel [Ergebnisse](#) stellt die gesammelten Ergebnisse, aufbereitet in Grafiken und als Code-Listings mit entsprechenden Interpretation, vor. Die Kapitel [Zusammenfassung](#) und [Ausblick](#) geben schließlich einen kurzen Rückblick der Arbeit wider und zeigen weitere Ausbaumöglichkeiten des entwickelten Programmcodes sowie Richtungen zur Forschung auf.

Chapter 2. Theoretische Grundlagen

Zunächst werden die theoretischen Grundlagen zum *Information Hiding* aufbereitet. Basierend darauf wird auf die Analyse von Merkmalen, die durch das Anwenden von Information Hiding entstehen, eingegangen.

2.1. Information Hiding

Die Methodik des Information Hiding beschreibt allgemein das Verstecken von Daten in anderen Daten, wobei in das *Cover*-Medium Daten eingebettet werden, wodurch das veränderte Medium mit eingebetteten Daten (allgemein als *Stego*-Medium bekannt) entsteht. Die eingebetteten Daten werden je nach Kontext als *Einbettungsdaten*, *Nachricht* oder auch *Payload* bezeichnet. Zum Information Hiding werden die Techniken der *Steganografie*, des *Watermarkings* (im weiteren Text als Wasserzeichen benannt) und die *Kryptografie* zugeteilt. Die Kryptografie wird in dieser Arbeit nicht weiter als eigenständiges Verfahren beleuchtet, da dabei nicht versucht wird, das originale Medium scheinbar unverändert zu lassen. [6, 7]

Neben den genannten Techniken fällt auch das Einbetten in die Struktur des Cover-Mediums zum Information Hiding, wobei zumeist Metadaten überschrieben oder Einbettungsdaten einfach an festgelegte Stellen im Medium geschrieben werden.

2.1.1. Anwendungsgebiete

Das Einbetten in die Struktur oder die Metadaten erfolgt im Allgemeinen spezifisch auf das Format angepasst. In ein Bild im JFIF-Format [8] lassen sich beispielsweise Daten an Stelle eines Vorschaubildes (sogenanntes *Thumbnail*) einbetten. Außerdem zählen zu dieser Art von Information Hiding das Anfügen von Daten nach einem End-Marker des Dateiformats wie dem EOI-Marker bei JPEG. Die oftmals simple Natur dieser Verfahren erlaubt eine schnelle Implementierung in Programmcode und ermöglicht meistens eine hohe Kapazität an Einbettungsdaten.

TODO:

- Quellen

Bei der Steganografie liegt der Fokus auf dem effektiven Verstecken der Existenz von eingebetteten Daten, sodass nur der Sendende und der Empfangende von der Nachricht weiß und diese auch nur entschlüsseln kann. Wie bereits in der Einleitung beschrieben, wurde diese Technik schon lange vor digitalen Systemen eingesetzt. [7]

Wasserzeichen weisen dagegen ein andere Anwendungsgebiete auf. Sie werden für das Fingerprinting, den Urheberrechtsschutz, die Verifikation der Authentizität von Inhalten (z.B. auf Social Media-Plattformen), in medizinischen Anwendungen und viele mehr angewandt. Daraus hervorgehend müssen Wasserzeichen anderen Kriterien genügen, als es Steganografie- oder Struktur-Verfahren tun. [9]

2.1.2. Kriterien

Die Techniken des Information Hiding werden nach drei Kriterien eingeordnet:

Kapazität	Beschreibt, welche Größe an Daten in das Cover eingebettet werden kann
Robustheit	Beschreibt, wie sicher das Verfahren vor Angriffen durch Dritte ist
Erkennbarkeit	Beschreibt, wie einfach die Verwendung des Verfahrens auf das Cover durch Dritte zu entdecken ist

Im folgenden Diagramm wird die Gewichtung von Struktur-Einbettungen, Steganografie und Wasserzeichen visualisiert. Ein Wert von 1 entspricht *niedriger*, 2 entspricht *mittlerer* und 3 entspricht *hoher* Kapazität und Robustheit, während ein Wert von 1 eine *hohe*, 2 eine *mittlere* und 3 eine *geringe* Erkennbarkeit widerspiegelt. Die Abbildung zeigt eine grobe Tendenz und ist nicht durch empirische Forschung konkret belegt.

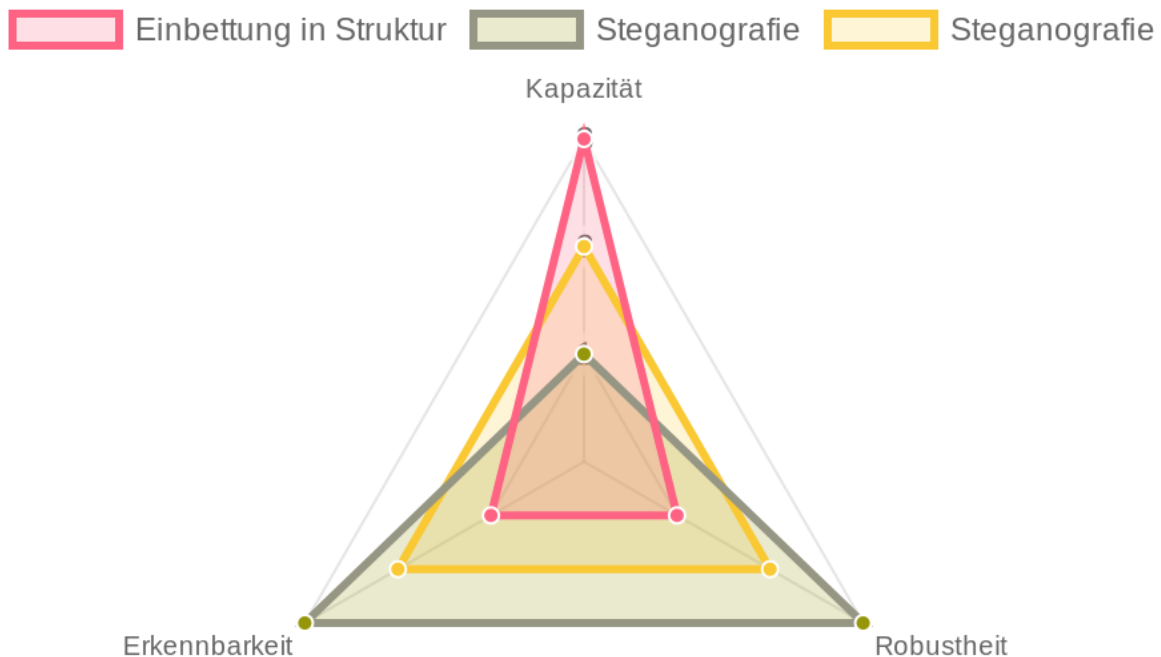


Abbildung 1. Gewichtung von Verfahren im Bereich der Struktur-Einbettung, Steganografie und Wasserzeichen

TODO:

- Quellen

Dabei ist zu erkennen, dass bei Struktur-Einbettungen mehr Wert darauf gelegt wird, größere Datenmengen verschleiert transportieren zu können, wofür die Robustheit und Erkennbarkeit zurückgestellt wird. Steganografie wird dagegen eingesetzt, um eine bessere Robustheit und niedrigere Erkennbarkeit zu erzielen. Wasserzeichen werden mit möglichst hoher Robustheit erstellt, wobei auch die Erkennbarkeit eine hohe Priorisierung hat, und die Kapazität eine untergeordnete Rolle spielt. Dies ist darin begründet, dass Wasserzeichen üblicherweise nur Identifikationsinformationen beinhalten.

2.1.3. Verfahren

Die klassischen Verfahren der Stegografie können in zwei Domänen unterteilt werden: der *Raum-* und der *Frequenz-*Domäne. In der Raumdomäne werden die Pixel des Cover-Bildes so verändert, dass die Nachricht darin versteckt ist und wieder extrahiert werden kann. In der Frequenz-Domäne werden die Komprimierungstabellen des Bildes genutzt, um durch gezielte Manipulation der darin enthaltenen Werte eine Nachricht zu verstecken. Neben den klassischen Verfahren werden zudem vermehrt auch neuronale Netze insbesondere für Wasserzeichen eingesetzt.

Die wichtigsten Verfahren werden nachfolgend kurz vorgestellt, ohne einen Anspruch auf Vollständigkeit zu erheben:

Raumdomäne

Least Significant Bit (LSB)

Bei dieser Technik wird der am wenigsten wichtige Bit eines Pixel in den Wert eines Bits aus der einzubettenden Nachricht geändert. Die Veränderung ist für das menschliche Auge minimal und daher durch reine Observation durch Menschen kaum zu entdecken. Neben des am wenigsten relevanten Bits kann auch der zweitwenigst relevante und der drittwenigst und so weiter zur Einbettung genutzt werden, wobei sich die Kapazität bezüglich der Einbettungslänge für eine immer leichtere Detektierbarkeit erhöht. Bei Farbbildern kann die Technik auf jeden der drei Farbkanäle sowie auf den Alpha-Kanal angewendet werden.

Intermedia Significant Bit (ISB)

Diese Verbesserung der LSB-Methode sucht pro Pixel in einem bestimmten Bit-Intervall nach dem optimalen Bit, dessen Veränderung am wenigsten Einfluss auf die Detektierbarkeit ausüben dürfte. Varianten dieser Methode passen die restlichen Bits außerdem so an, dass der Farbwert des Pixels am dem ursprünglichen entspricht.

Patchwork

Diese Technik nutzt eine pseudo-zufällige Einbettung von sich wiederholenden Mustern mittels einer Gauss-Verteilung. Dabei werden zwei Patches A und B pseudo-zufällig ausgewählt, wobei A verblasst und B verdunkelt wird.

Frequenzdomäne

Discrete Cosine Transform (DCT)

Die DCT-Methode nutzt die DCT-Tabelle von JPEG-Bildern, indem meistens das Cover-Bild in mehrere Blöcke unterteilt wird, auf die die Einbettung ausgeführt wird. Üblicherweise werden die niederen Frequenzen zur Manipulation genutzt, da diese durch einfachste Verfahren wie Sichtprüfung durch Menschen schwerer detektierbar sind.

Discrete Wavelet Transform (DWT)

Die Frequenzen des Bilds werden in sogenannte Wavelets umgewandelt, welche für die Signalverarbeitung oder Bildkomprimierung relevant sind. Die dabei verwendeten Koeffizienten können in mehrere Ebenen unterteilt werden, welche verschiedene Manipulationen erlauben.

Singular Value Decomposition (SVD)

Verfahren der SVD-Technik nutzen Eigenschaften der Eigendekomposition einer symmetrischen

Matrix mit nicht-negativen Eigenwerten sowie deren Beziehungen zu den Koeffizienten.

Weitere Techniken werden insbesondere in der Forschung neu- und weiterentwickelt. Auch Hybrid-Verfahren der beiden Domänen sind möglich. In dieser Arbeit soll allerdings nur ein Überblick über die gängigsten Techniken verschafft werden. Einen guten Einblick in das breitere Forschungsfeld und vertiefende Informationen zu den Kurzbeschreibungen liefern Begum und Uddin in [9].

Zum Bereich des Information Hiding gehört auch die Einbettung in die Struktur des Cover-Bildes, z.B. die folgenden Methoden:

Struktur-Verfahren

Einbetten in Metadaten

Hierbei wird die Nachricht in ein Feld der Metadaten des Cover-Bildes geschrieben. Dabei kann es sich um das Kommentar-Feld oder auch das Model-Feld in den EXIF-Metadaten handeln. Für größere Nachrichten wie Payloads muss jedoch insbesondere bei EXIF-Feldern der durch Bildverarbeitungsprogramme erwartete Datentyp beachtet werden.

Anfügen ans Dateiende

Die Nachricht wird nach dem End-Marker der Bilddatei eingefügt. Die meisten Bilddarstellungsprogramme ignorieren alle weiteren Anhänge, wodurch ein Payload für einen menschlichen Betrachter nicht sichtbar wird, jedoch durch Forensik-Programme einfach zu erkennen ist.

Ändern des Vorschaubilds

Eine Nachricht kann an Stelle eines Vorschaubilds in das Cover-Bild eingefügt werden. Eine einfache Detektion durch Sicht kann dann nur erfolgen, falls das Vorschaubild dargestellt werden soll.

Struktur-Verfahren können noch in vielen weiteren Variationen eingesetzt werden. Außerdem können Struktur-Verfahren mit Raum- und Frequenz-Verfahren kombiniert werden, um beispielsweise die Kapazität zu erhöhen.

2.2. Merkmalsanalyse

Die *Merkmalsanalyse* im Kontext dieser Arbeit beschäftigt sich mit der Suche nach und der Analyse von gefundenen Merkmalen von vermutlichen steganografisch manipulierten Daten. Damit ist sie der sogenannten *Attribution* zugehörig, welche sich grundsätzlich mit dem Ziel beschäftigt, den oder die Verursachenden von böswilliger Cyberaktivität wie zum Beispiel das Eindringen in Rechnersysteme. Dabei kann es zunächst um die Rückverfolgung auf des Angriffs auf die durch die Angreifer verwendeten Systeme gehen, welche wiederum Rückschlüsse auf die Angreifer selbst ermöglichen. Oftmals werden Angriffe im Auftrag einer Drittpartei durchgeführt, die möglicherweise über die Angreifer identifizierbar sind. [10]

TODO:

- mehr zu Politik und Gesellschaft

Die Merkmalsanalyse in dieser Arbeit fokussiert sich nur auf die Identifikation von Programmen,

die für die Einbettung von Daten mittels Information Hiding oder im Speziellen Steganografie genutzt wurden, was auch als *Steganalyse* bezeichnet wird. Eine solche Merkmalsanalyse wird durchgeführt, nachdem ein erhärterter Verdacht auf die Nutzung von Steganografie durch vorherige Schritte der Steganalyse festgestellt wurde. Je nach erwartetem Steganografie-Verfahren werden unterschiedliche Analyse-Verfahren angewendet und zumeist miteinander kombiniert, um die Einbettungssoftware so detailliert wie möglich zu identifizieren. Konkrete Methoden werden in den Ergebnissen vorgestellt.

Chapter 3. Methodik

Die folgenden Methoden wurden zum Erreichen der [Zielsetzung](#) durchgeführt:

- *Literaturrecherche*: Um die Ziele **Z.1**, **Z.2** und **Z.3** zu erfüllen, wurde diese Methodik genutzt. Hauptsächlich wurden bei den Literaturrecherchen neben wissenschaftlichen Papern besonders auf technische Berichte von renomierten Akteuren im Bereich der IT-Sicherheit und speziell der IT-Forensik zurückgegriffen. Die Analyse von Online-Datenbanken für Malware erfolgte teilautomatisiert. Für die Suche nach geeigneter Software wurde aus den zuvor genannten Quellen weitere Verweise verfolgt sowie Suchen auf bekannten Software-Entwicklungsplattformen wie GitHub durchgeführt.
- *Software-Entwicklung*: Zur Erfüllung von **Z.4** wurde diese Methodik gewählt, da keine der gefundenen Tools entsprechende Funktionen bietet. Die Entwicklung fand nach dem Top-Down-Prinzip mit dem Test-Driven-Prinzip statt. Die Tests wurden direkt auf die Forschungsdaten angewendet und lassen sich automatisiert mit diesen ausführen.
- *Experimentelle Validierung*:

3.1. Suchmaschinen

Die manuellen Recherchen wurde über die folgenden Suchmaschinen durchgeführt, falls nicht anders beschrieben:

- *scibo*: Die interne Suchmaschine der Technischen Hochschule Brandenburg, welche den sogenannten EBSCO Discovery Service nutzt
- *Springer Link*: Eine Suchmaschine des Springer Verlags für wissenschaftliche Publikationen ihres Verlags
- *Springer Professional*: Weitere Suchmaschine des Springer Verlags für wissenschaftliche Publikationen ihres Verlags
- *IEEE Xplore*: Bietet renomierten Zugang zu wissenschaftlichen Inhalten, die durch das Institut veröffentlicht wurden
- *arxiv*: Ein freier Verteilungsdienst für wissenschaftliche Artikel, der jedoch selbst keine Peer-Reviews durchführt
- *Google*: Die Suchmaschine der gleichnamigen Firma zur Ausweitung der Suche über wissenschaftliche Publikationen hinaus

3.2. Literaturrecherche zu Malware-Vorkommnissen im Zusammenhang mit Bild-Steganografie

Die Recherche erfolgte in zwei Schritten:

- Schaffen einer Übersicht zu Malware-Vorkommnissen aus wissenschaftlichen Publikationen und technischen Berichten
- Automatisierte Sammlung von Malware-Vorkommnissen aus öffentlichen Datenbanken (ohne Malware-Samples zu analysieren)

3.2.1. Manuelle Recherche

Die folgenden Suchbegriffe wurden in den zuvor genannten [Suchmaschinen](#) verwendet: *"Malware steganography"*, *"stego malware report"*, *"steganography malware analysis report 2023"*, *"stego malware"*, *"steganography malware report"*, *"steganography malware analysis"*, *"steganography malware survey"* und *"steganography malware database"*. Die Suchanfragen wurden so gewählt, um gezielt nach Berichten zu suchen, die allgemeine Analysen liefern, statt auf einzelne Malware fokussiert zu sein. Im wissenschaftlichen Bereich wurden hauptsächlich Artikel zur Analyse von Bild-Steganografie mittels Künstlicher Intelligenz, im Bereich von Blockchains oder der Prävention im IT-Sicherheitsmanagement gefunden.

Jedoch gerade aus den letzten drei Jahren ließen sich zwei Paper finden, die eine Urhebung oder Sammlung an Malware-Vorkommnissen mit Bild-Steganografie vorstellen. Da kaum wissenschaftliche Artikel in den Jahren zuvor zu diesem Thema publiziert wurden, lässt sich ein Trend bei Malware-Autoren und auch Forschung hin zu Steganografie-Malware erkennen. In [\[11\]](#) werden 10 Malware, die Steganografie oder Information Hiding verwenden, vorgestellt. Der Artikel verweist zudem auf ein Repository namens *steg-in-the-wild* von einem der Autoren des Artikels, in dem eine aktualisierte Liste der Malware geführt wird, welche im weiteren Verlauf der Recherche genutzt wurde. [\[12\]](#)

Des Weiteren wurde in [\[13\]](#) 16 Malware benannt, die sich zum Teil mit der Arbeit von Caviglione und Wojciech decken.

Im technischen Bericht [\[14\]](#) werden sechs Malware und die konkrete Funktionsweise jener an einem Beispiel beschrieben. Darin wie auch in [\[15\]](#) wird die *MITRE Att&ck*-Wissenbasis referenziert, in der eine systematische Zuordnung von sogenannten Techniken und Taktiken, die von Malware eingesetzt werden, zu Malware-Vorkommnissen durchgeführt wird. So wird unter den Techniken *T1001.002: Data Obfuscation: Steganography* und *T1027.003: Obfuscated Files or Information: Steganography* der Einsatz von Steganografie bei Malware-Vorkommnissen dokumentiert. [\[16\]](#)

Eine weitere Datenbank für Malware-Vorkommnisse führt das Fraunhofer FKIE mit malpedia, die eine Suchmaschine zum Finden von Malware nach Suchbegriffen im Namen oder der Beschreibung bietet. [\[17\]](#)

3.2.2. Automatisierte Sammlung aus Datenbanken

Um eine umfängliche Übersicht zu schaffen, wurden daraufhin die Datenbanken von MITRE und Malpedia sowie die Artikelsammlung in *steg-in-the-wild* ausgewählt. Nach der manuellen Recherche sollte eine Übersicht geschaffen werden, die auch weniger beachtete Malware-Vorkommnisse einbezieht, um so eine höhere statistische Auswertbarkeit für die Ergebnissliste zu erzielen.

Die Automatisierung wurde mit Hilfe eines Jupyter Notebooks durchgeführt. Dabei handelt es sich um eine interaktive, lokal ausführbare Plattform, die es erlaubt, Datenanalysen und -visualisierung mit direktem Feedback mittels Python oder anderen Programmiersprachen durchzuführen. Die folgenden Python-Bibliotheken wurden neben der Standard-Bibliothek zusätzlich verwendet:

- *Selenium*: Ermöglicht das automatische Öffnen von und Interagieren mit Webseiten über einen

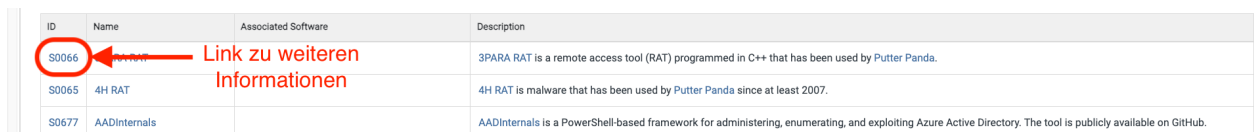
lokalen Webbrowser

- *Pandas*: Ermöglicht eine verbesserte Datenverarbeitung von tabularischen Daten
- *requests*: Bietet ein einfaches Interface zum Herunterladen von Daten
- *bibtexparser*: Ermöglicht das Parsen von BibTex-Dateien

Außerdem wurde die Erweiterung *jupyter-ai* für Jupyter genutzt, um generative Text-KI für die initiale Generierung eines sogenannten Web-Scrapers zu nutzen. Die Aufgabe des Web-Scrapers ist das Sammeln (oder Schürfen) von Informationen aus Webdaten, vornehmlich HTML-Dateien.

MITRE-Datenbank

In Abbildung 1 sind Teile der Software-Tabelle der MITRE-Datenbank zu sehen, deren Links nun gesammelt und nach Steganografie-Malware gefiltert werden soll.



ID	Name	Associated Software	Description
S0066	3PARA RAT		3PARA RAT is a remote access tool (RAT) programmed in C++ that has been used by Putter Panda.
S0065	4H RAT		4H RAT is malware that has been used by Putter Panda since at least 2007.
S0677	AADInternals		AADInternals is a PowerShell-based framework for administering, enumerating, and exploiting Azure Active Directory. The tool is publicly available on GitHub.

Abbildung 2. Software-Tabelle der MITRE-Datenbank [18]

Im Folgenden wird ein Code-Block dargestellt, in dem ein Web-Scraper automatisiert die zuvor genannte Tabelle nach den relevanten Daten durchforstet.

```
1 import os
2 from selenium import webdriver
3 from selenium.webdriver.chrome.service import Service
4 from selenium.webdriver.common.by import By
5 from selenium.webdriver.support.ui import WebDriverWait
6 from selenium.webdriver.support import expected_conditions as EC
7
8 # Get the path to chromedriver from environment variable
9 chrome_driver_path = os.environ.get('CHROMEDRIVER_PATH')
10
11 # Set up Chrome driver ①
12 options = webdriver.ChromeOptions()
13 options.add_argument('--headless') # Run Chrome in headless mode
14 options.add_argument('--disable-extensions')
15 options.add_argument('--disable-dev-shm-usage')
16 service = Service(chrome_driver_path)
17 driver = webdriver.Chrome(service=service, options=options)
18
19 FOUND_LINKS_FILE = 'data/mitre-attack-stego-malware.txt'
20
21
22 def search_mitre_attack(mitre_attack_url='https://attack.mitre.org/software/'):
23     global driver
24     # Open the link
25     driver.get(mitre_attack_url)
26
27     # Collect links in the first column of the table ②
```

```

28     links = driver.find_elements(By.CSS_SELECTOR, 'table tr td:nth-child(1) a')
29     link_urls = [link.get_attribute('href') for link in links]
30
31     driver.quit()
32
33     # Iterate over each link
34     found_links = []
35     for link_url in link_urls:
36         driver = webdriver.Chrome(service=service, options=options)
37         try:
38             # Open the link
39             driver.get(link_url)
40
41             # Wait for the page to fully load ③
42             WebDriverWait(driver, 10).until(
43                 EC.presence_of_element_located((By.TAG_NAME, 'body'))
44             )
45
46             # Search for the terms "steganography" in the page ④
47             page_content = driver.page_source.lower()
48             if 'steganography' in page_content:
49                 found_links.append(link_url)
50         except Exception as e:
51             print(f'Error occurred while processing link: {link_url}')
52             print(e)
53         finally:
54             # Quit the driver
55             driver.quit()
56     return found_links

```

① Initialisierung des Chrome-Webdrivers

② Selektion der Tabelle mit `table` → `tr` (Tabellenzeilen) → `td:nth-child(1)` (aus den Tabellendaten jeweils die erste Zelle) → `a` (das HTML-Element für Links)

③ Wartet, bis die Seite geladen wurde

④ Sucht im Text der Webseite in Kleinbuchstaben nach `steganography`

Die Zeilen 12 bis 17 initialisieren den Chrome-Webdriver, der für die Kommunikation mit dem lokalen Chrome-Browser genutzt wird. Chrome wird *headless* (ohne Benutzeroberfläche), ohne Erweiterungen und mit der Option, Arbeitsspeicher auf die Festplatte auszulagern, gestartet.

Der Block von Zeile 22 bis 56 definiert eine neue Funktion `search_mitre_attack(mitre_attack_url)`. Darin wird zunächst mittels des Webrivers der Link zur Software-Seite von MITRE Att&ck geöffnet, dann in den beiden Zeilen 28 und 29 die Tabelle mit den allen gelisteten Softwares über einen CSS-Selektor ausgewählt und die gesammelten Links aus den `href`-HTML-Attributen extrahiert.

Es wird über die Links iteriert, gewartet, bis der Hauptteil der Seite im Browser geladen wurde, und der Seiteninhalt nach dem Begriff *steganography* durchsucht. Konnte der Begriff gefunden werden, wird der Link einer Ergebnisliste hinzugefügt. Der Webdriver wird bei jedem Durchlauf neu geöffnet, da es sonst zu massiven Speicher-Leaks kommt und der Chrome-Prozess bis zu einem

Crash Speicher konsumiert.

Abbildung 2 zeigt ausschnittsweise, wie Software auf der MITRE-Webseite angezeigt wird. Markiert sind die Datenpunkte, die extrahiert werden sollen.



Abbildung 3. Beispiel einer Software auf der MITRE-Webseite [19]

Im nächste Code-Block wird der dazu verwendete Web-Scraper vorgestellt.

```
1 def scrape_mitre_attack_software(link):
2     driver = webdriver.Chrome(service=service, options=options)
3     driver.get(link)
4
5     # Search for name in h1 tag ①
6     name = driver.find_element(By.TAG_NAME, 'h1').text
7     # Search for description in tag with class 'description-body' ②
8     description = driver.find_element(By.CLASS_NAME, 'description-body').text
9
10    # Search for card body ③
11    card_body = driver.find_element(By.CLASS_NAME, 'card-body')
12    card_data_divs = card_body.find_elements(By.CLASS_NAME, 'card-data')
13
14    card_data_dict = {
15        'Name': name,
16        'Description': description,
17    }
18    # Iterate card elements and extract key-value pairs ④
19    for card_data in card_data_divs:
20        col_md_11_div = card_data.find_element(By.CLASS_NAME, 'col-md-11')
21        key, value = col_md_11_div.text.split(':')
22        card_data_dict[key] = value
23
24    # Locate 'Techniques Used' table ⑤
25    techniques_table = driver.find_element(By.CLASS_NAME, 'techniques-used')
26    techniques_rows = techniques_table.find_elements(By.TAG_NAME, 'tr')
27
28    # Collect techniques table header ⑥
29    techniques_headers = [
30        header.text
31        for header in techniques_rows[0].find_elements(By.TAG_NAME, 'th')
32    ]
33    techniques = []
34    for row in techniques_rows[1:]: ⑦
35        row_classes = row.get_attribute('class')
```

```

36     technique_dict = {}
37
38     # Get table columns
39     columns = row.find_elements(By.TAG_NAME, 'td')
40
41     if len(columns) == 4:
42         # In case there are only 4 columns, we can put them as-is in the
dictionary
43         for index, header in enumerate(techniques_headers):
44             technique_dict[header] = columns[index].text
45     elif 'noparent' in row_classes:
46         # If the 'noparent' class is present, we expect 5 columns and merge the
2nd and 3rd into one
47         technique_dict[techniques_headers[0]] = columns[0].text
48         technique_dict[techniques_headers[1]] = ''.join([columns[1].text,
columns[2].text])
49         technique_dict[techniques_headers[2]] = columns[3].text
50         technique_dict[techniques_headers[3]] = columns[4].text
51     else:
52         # Otherwise we suspect this entry to be a sub-technique.
53         # In this case, we copy the name and description of the previous
technique
54         prev_technique = techniques[-1]
55         technique_dict[techniques_headers[0]] = prev_technique
[techniques_headers[0]]
56         technique_dict[techniques_headers[1]] = prev_technique
[techniques_headers[1]]
57         technique_dict[techniques_headers[2]] = columns[3].text
58         technique_dict[techniques_headers[3]] = columns[4].text
59         techniques.append(technique_dict)
60
61     driver.quit()
62
63     card_data_dict['MITRE ID'] = card_data_dict['ID'] ⑧
64     del card_data_dict['ID']
65
66     return {
67         **card_data_dict,
68         'Techniques Used': techniques,
69     }

```

- ① Suche nach HTML-Element `h1` für den Name der Technik
- ② Suche nach HTML-Element mit der CSS-Klasse `description-body`, welches die Beschreibung der Technik beinhaltet
- ③ Sammle im HTML-Element mit der CSS-Klasse `card-body` alle Unterelemente mit der Klasse `card-data` ein
- ④ Extrahiere aus den Kartenelementen jeweils ein HTML-Element mit der Klasse `col-md-11`, welches einen Schlüssel und Wert (Variable `key` und `value`), separiert von `:`, beinhaltet

- ⑤ Finde die *Techniques Used*-Tabelle anhand der CSS-Klasse `techniques-used` und sammle die Tabellenzeilen ein
- ⑥ Extrahiere den Tabellenkopf aus der ersten Zeile der Tabelle
- ⑦ Iteriere über die verbleibenden Tabellenzeilen und extrahiere die Informationen zur jeweiligen Technik nach Bedingungen (genau beschrieben im Text)
- ⑧ Ersetze `ID`-Schlüssel mit `MITRE ID`, um die Herkunft der ID im Datensatz zu signalisieren

Wie im Web-Scraper zuvor wird in diesem die Webdriver bei jedem Durchlauf in der zweiten Zeile reinitialisiert. Daraufhin wird der Name und die Beschreibung der MITRE Software in den Zeilen 6 und 8 extrahiert, in dem nach der Überschrift der Seite (`h1`) und einem HTML-Element mit der CSS-Klasse `description-body` gesucht wird. Als nächstes wird die Karte, wie in Abbildung 2 zu sehen, herausgesucht und die einzelnen Schlüssel-Wert-Paare extrahiert. Die Paare werden beim Doppelpunkt getrennt und als Schlüssel und Wert im `card_data_dict`-Dictionary gespeichert. Es folgt die Extraktion der genutzten Techniken aus der Tabelle *Techniques Used*. Dazu wird die Tabelle mit der CSS-Klasse `techniques-used` selektiert, der Tabellenkopf aus der ersten Zeile entnommen und die restlichen Zeilen iteriert.

Die Abbildung 3 zeigt in einem Auszug, wie die Tabellen aussehen können.

Techniques Used CSS: noparent

Domain	ID	Name	Use
Enterprise	T1134	Access Token Manipulation	Sliver has t
Enterprise	T1071	Application Layer Protocol: Web Protocols	Sliver has t
		.001	
		.004	Sliver can s
Enterprise	T1132	Data Encoding: Standard Encoding	Sliver can u
Enterprise	T1001	Data Obfuscation: Steganography	Sliver can e
Enterprise	T1573	Encrypted Channel: Symmetric Cryptography	Sliver can u
		.002	Sliver can u
Enterprise	T1041	Exfiltration Over C2 Channel	Sliver can e

CSS: sub

Abbildung 4. Beispiel einer *Techniques Used*-Tabelle [19]

Die *ID*-Spalte ist teilweise in zwei Unterspalten unterteilt. Dies ist bei der Datensammlung in den Zeilen 41 bis 58 berücksichtigt. Daraus ergeben sich die folgenden Bedingungen und Abarbeitungen:

1. Es sind insgesamt 4 Spalten \Rightarrow Übernahme die Zellen entsprechend des Tabellenkopfs
2. Die Zeile hat die CSS-Klasse `noparent` und es sind 4 Spalten mit einer geteilt in zwei \Rightarrow Füge die zweite und dritte Zelle zusammen als zweite Zelle, übernehme die restlichen Zellen

3. Die Zeile beinhaltet die CSS-Klasse `sub` und die ersten beiden Spalten sind leer \Rightarrow Übernahme die Daten der ersten beiden Zellen aus der vorherigen Zeile, übernehme die restlichen Zellen

Abschließend wird die `ID` in `MITRE ID` umbenannt, um die Herkunft im Datensatz klarzustellen und das Dictionary mit den Techniken zusammengeführt.

Anzumerken ist dabei, dass die beiden Web-Scraper in einer optimierten Version zusammengefasst werden sollten, sodass die Software-Seiten nicht zwei Mal geöffnet und nach relevanten Informationen durchsucht werden müssen. Bei der iterativen Arbeit mit dem Jupyter Notebook war die geteilte Vorgehensweise allerdings leichter zu testen.

Malpedia-Datenbank

Im folgenden Code-Block wird ein weitere Funktion vorgestellt, die einen allgemeinen Ansatz zum Web-Scraping verfolgt und nicht speziell auf eine Webseite zugeschnitten ist. Dies wurde unter anderem genutzt, um nach Steganografie-Malware im Bildbereich in der Malware-Datenbank *Malpedia* des Fraunhofer FKIE zu suchen.

```
1 def scrape_malware_data(url, name=None, description=None, created_at=None):
2     data = None
3
4     driver = webdriver.Chrome(service=service, options=options) ①
5     try:
6         driver.get(url)
7
8         # Wait for the page to fully load
9         driver.implicitly_wait(10)
10
11        page_content = driver.page_source.lower() ②
12        if 'steganography' in page_content:
13            return data
14
15        try:
16            # Look for the title in the meta tags ③
17            name = driver.find_element(By.XPATH, '//meta[@name="title" or
18@property="og:title"]').get_attribute(
19                'content')
20        except NoSuchElementException:
21            pass
22
23        # The description is not always available
24        try:
25            # Look for a meta tag that contains "description" in the name or
26            # property attribute ④
27            description = driver.find_element(By.XPATH,
28                '//meta[@name="description" or
29contains(@property, "description")]').get_attribute(
30                'content')
31        except NoSuchElementException:
32            pass
```

```

30
31     # The creation date is not always available
32     try:
33         # Look for a meta tag that contains "created" or "published" in the
name or property attribute ⑤
34         created_at = driver.find_element(By.XPATH,
35                                         '//meta[contains(@name, "created") or
contains(@name, "published") or contains(@property, "created") or
contains(@property, "published")]').get_attribute(
36                                         'content')
37     except NoSuchElementException:
38         pass
39
40     platforms = [platform for platform in PLATFORMS if platform.lower() in
page_content] ⑥
41     data = {
42         'Name': name,
43         'Description': description,
44         'Type': 'MALWARE',
45         'Created': created_at,
46         'Platforms': platforms,
47         'References': [url],
48     }
49 except Exception as e:
50     print(f'Error occurred while processing link: {url}')
51     print(e)
52 finally:
53     driver.quit()
54
55     return data

```

- ① Initialisiere des Webdriver bei jedem Aufruf (wie in vorherigen Web-Scrapern)
- ② Sucht nach dem Wort `steganography` im Text der Webseite in Kleinbuchstaben
- ③ Sucht nach einem `meta`-HTML-Element mit dem Attribut `name` gesetzt auf `title` **oder** dem Attribut `property` gesetzt auf `og:title`
- ④ Sucht nach einem `meta`-HTML-Element mit dem Attribut `name` gesetzt auf `description` **oder** dem Attribut `property`, dessen Wert `description` enthält
- ⑤ Sucht nach einem `meta`-HTML-Element mit dem Attribut `name`, dessen Wert `created` oder `published` enthält, **oder** dem Attribut `property`, dessen Wert `created` oder `published` enthält
- ⑥ Sucht nach den Namen von Plattformen, die in der Konstante `PLATFORMS` definiert sind, die als Ziele der auf der Webseite beschriebenen Malware gewählt worden sein könnten

Der Ablauf des generischen Web-Scrapers ist ähnlich zu den vorherigen. Der Webdriver wird immer neu initialisiert. Es wird nach dem Wort `steganography` im Text der Webseite in Kleinbuchstaben gesucht. Danach wird versucht, Meta-Informationen aus der Webseite wie den Namen des Artikels, eine Beschreibung dazu, das Erstellungs- und Änderungsdatum zu extrahieren. Dabei werden XPath-Selektoren verwendet, da diese sehr flexibel mit XML- bzw. HTML-Elementen und deren Attributen arbeiten können. Außerdem wird versucht, durch die in dem Artikel

beschriebene Malware angegriffene Plattformen zu finden.

Die gesammelten Daten werden in einem Dictionary zusammengeführt, das in der Struktur den MITRE-Daten gleicht, um eine einfache Zusammenführung und Weiterverarbeitung zu ermöglichen.

Im Fall der Malpedia-Datenbank können alle Referenzen zu den dort enthaltenen Einträgen als BibTex-Datei unter <https://malpedia.caad.fkie.fraunhofer.de/library/download> heruntergeladen werden. Der nächste Code-Block zeigt die Sammlung der Links aus den BibTex-Einträgen, wobei in den Titeln nach **steganography** gesucht wird. Dies ist nicht die umfassendste Methode, ermöglicht aber einen Überblick in kürzerer Zeit zu gewinnen, als jeden einzelnen der 15069 Einträge (zum Zeitpunkt der letzten Ausführung am 23.02.24) mit dem Web-Scraper durchzugehen.

```
1 import bibtexparser
2
3 bibliography = bibtexparser.parse_file(MALPEDIA_BIBLIOGRAPHY_FILE)
4
5 stego_malware_entries = []
6 for entry in bibliography.entries:
7     if 'steganography' in entry['title'].lower():
8         stego_malware_entries.append(entry)
```

Die so vorgefilterten Einträge wurden dann mit dem Webscraper **scrape_malware_data** weiter untersucht.

stego-in-the-wild

Als letzte Datenquelle wurde das Repository <https://github.com/lucacav/steg-in-the-wild> von Luca Caviglione verwendet und die relevanten Daten aus der **README.md**-Datei gesammelt, im folgenden Code-Block zu sehen.

```
1 def extract_links_from_list(text):
2     lines = text.splitlines()
3
4     # We only want the first bullet list ①
5     list_entries = itertools.dropwhile(lambda line: not line.startswith('*'),
6     lines)
7     list_entries = itertools.takewhile(lambda line: line.startswith('*'),
8     list_entries)
9
10    # Extract links and their descriptions ②
11    links = []
12    for entry in list_entries:
13        link, description = entry.split(':', 1)
14        name, url = link.split('](', 1)
15        name = name[3:]
16        url = url[:-1]
17        links.append((name, url, description.strip()))
```


- ① Sammle die erste Stichpunktliste ein und ignoriere alle weiteren
- ② Zerlege die Einträge am Doppelpunkt, um den Namen, den Link und zugehörige Beschreibung zu extrahieren

Die Ergebnisliste aus Tupeln mit dem Namen des Artikels, dem zugehörigen Link und der Beschreibung des Repository-Autors wurde wiederum elementweise als Parameter an den Webscraper `scrape_malware_data` übergeben.

Automatisierte Datenzusammenführung und -säuberung

Im letzten Schritt wurden die Ergebnislisten zusammengeführt und bereinigt. Dabei wurden die Daten aus der MITRE-Datenbank als Ausgangspunkt genommen.

```

1 malpedia_malware_data = [malware for malware in malpedia_malware_data if malware is
  not None]
2
3 processed_malware_data = mitre_attack_data
4
5 # Try to extract any malware names from the other datasets than MITRE Att@ck and
  add them to the list ①
6 def try_add_malware_data(data):
7     new_malware_data = []
8     for entry in data:
9         name = entry['Name']
10        split_name = name.split(':')
11        if len(split_name) > 1:
12            name = split_name[0].strip()
13            entry['Name'] = name
14            new_malware_data.append(entry)
15            data.remove(entry)
16    return new_malware_data
17
18 processed_malware_data += try_add_malware_data(malpedia_malware_data)
19 processed_malware_data += try_add_malware_data(sitw_data)
20
21 # Try to match the processed malware data with the left over Malpedia and steg-in-
  the-wild data
22 def compare_names(name, other_name):
23     return name in other_name or name.replace(' ', '') in other_name or name
  .replace('-', '') in other_name
24
25 for entry in processed_malware_data: #②
26     name = entry['Name'].lower()
27     for malpedia_entry in malpedia_malware_data:
28         malpedia_name = malpedia_entry['Name'].lower()
29         if compare_names(name, malpedia_name):
30             entry['References'] = entry.get('References', []) + malpedia_entry.get(
31                 'References', [])

```

```

32     malpedia_malware_data.remove(malpedia_entry)
33
34     for sitw_data_entry in sitw_data:
35         sitw_name = sitw_data_entry['Name'].lower()
36         if compare_names(name, sitw_name):
37             entry['References'] = entry.get('References', []) + sitw_data_entry.
get(
38                 'References', [])
39             sitw_data.remove(sitw_data_entry)
40
41 # Convert the list of dictionaries to a DataFrame and clean the data
42 malware_data = pd.DataFrame(processed_malware_data + malpedia_malware_data +
sitw_data)
43
44 malware_data = malware_data[malware_data['Name'].str.contains('404') == False] ③
45 malware_data = malware_data[malware_data['Description'].str.contains('404') ==
False]
46 malware_data = malware_data[malware_data['Name'].str.contains('Not Found', case
=False) == False]
47 malware_data = malware_data[malware_data['Description'].str.contains('Not Found',
case=False) == False]
48
49 malware_data['Created'] = pd.to_datetime(malware_data['Created'], errors='coerce',
format='mixed', utc=True).dt.date ④
50
51 malware_data['Last Modified'] = pd.to_datetime(
52     malware_data['Last Modified'],
53     errors='coerce',
54     format='mixed',
55     utc=True).dt.date
56
57 malware_data['Platforms'] = malware_data['Platforms'].apply(
58     lambda platforms: ', '.join(platforms)
59     if isinstance(platforms, list) else platforms
60 ) ⑤
61
62 malware_data['Techniques Used'] = malware_data['Techniques Used'].apply(
63     lambda techniques: ', '.join(
64         technique['Use']
65         for technique in techniques
66         if 'Steganography' in technique['Name']
67     ) if techniques is not np.nan else None
68 ) ⑥
69
70 malware_data['References'] = malware_data['References'].apply(
71     lambda refs: ', '.join(refs)
72     if refs is not np.nan else None
73 ) ⑦
74
75 malware_data = malware_data.drop_duplicates(subset=['Name'], keep='first') ⑧

```

- ① Versucht, einen Malware-Namen aus einem Artikel-Namen zu extrahieren, indem alles vor einem Doppelpunkt als Malware-Name angenommen wird
- ② Fügt Einträge aus *Malpedia* und *steg-in-the-wild* als Referenzen zu MITRE-Einträgen hinzu, wenn der Artikel-Name den Malware-Namen beinhaltet
- ③ Entfernt alle Einträge, die **404** und **Not Found** im Namen oder der Beschreibung beinhaltet, da diese nicht valide sind
- ④ Säubert die Datumswerte
- ⑤ Fügt die Plattformen als Text, mit Kommata getrennt, zusammen
- ⑥ Fügt die Techniken als Text zusammen, die mit Steganografie zutun haben
- ⑦ Fügt die Referenzen als Text, mit Kommata getrennt, zusammen
- ⑧ Entfernte Duplikate

Es wird zunächst versucht, einen Malware-Namen aus den Artikel-Namen zu extrahieren, indem alles vor einem Doppelpunkt als Malware-Name angenommen wird. Dann werden die Einträge durchgegangen und Einträge mit Malware-Namen im Artikel-Namen werden dem Malware-Eintrag als Referenz beigelegt. Anschließend findet die Datenbereinigung statt.

Das finale Ergebnis wird im Abschnitt [Malware-Vorkommnisse im Zusammenhang mit Steganografie](#) ausgewertet, woraus die Anforderungen an den folgende Literaturrecherche abgeleitet wurden.

3.3. Literaturrecherche zu Steganografie- und Wasserzeichen-Tools

Basierend auf der zuvor erfolgten Sammlung an Malwares, welche Information Hiding Techniken mit Bilddaten nutzen, wurde die Recherche mit den nachfolgenden Kriterien durchgeführt.

1. **Bekanntheit:** Die Software muss in wissenschaftlichen Publikationen verwendet worden sein oder auf gängigen Software-Plattformen wie Github mindestens 1000 Bewertungen oder Downloads vorweisen können.
2. **Funktionsweise:** Der in der Software implementierte Algorithmus zum Information Hiding muss zumindest in einem Überblick dokumentiert sein oder zumindest der Quellcode muss verfügbar sein, damit die Software einer Kategorie entsprechend den [Theoretische Grundlagen](#) zugeordnet werden kann.
3. **noch was:** das

3.3.1. Steganografie-Software

Die Recherche wurde mit den folgenden Suchbegriffen durchgeführt: *"information hiding"*, *"information hiding tools"*, *"Steganography Detection"*, *"Steganography Techniques"*, *"steganography overview"*, *"steganography survey"*, *"steganography tools"* und *"steganography apps"*.

Die Suche im wissenschaftlichen Umfeld ergab insbesondere die Arbeiten von Pilania et al. in [20] und [21] sowie von Virma et al. in [22]. Es zeigen sich darin Überschneidungen in den gängigen

Tools, die zur Evaluierung und Detektion von Steganografie in Bildern genutzt werden. Diese Tools sind nur auf den Desktop-Plattformen ausführbar. Breuer [23] hat eine Software-Sammlung an Steganografie-Tools zusammengestellt, die sich ebenfalls vielfältig mit den in den Publikationen verwendeten Programmen deckt.

Im Bereich der mobilen Applikationen existieren ebenfalls Steganografie-Programme, welche im Folgenden als Steganografie-Apps bezeichnet werden, wie die Arbeiten von Chen et al. [24] und [25] sowie von Newman et al. [26] zeigen.

Aus der Recherche ging hervor, dass insbesondere Steganografie und zugehörige Desktop-Tools wie auch mobile Apps sowohl in der Wissenschaft als auch in der technischen Umsetzung Beachtung finden, während weitere Techniken des Information Hiding wie das Einbetten in Metadata oder andere selten betrachtet werden.

Die gefundenen Programme wurden zusammengetragen und in zwei Tabellen aufgeteilt, welche jeweils die Desktop-Tools und die mobilen Apps vorstellen. Diese sind in den Ergebnissen zu finden.

3.3.2. Wasserzeichen-Software

Wie in den [theoretischen Grundlagen zu Wasserzeichen](#) beschrieben werden Wasserzeichen entweder sichtbar oder unsichtbar eingebettet, wobei nur die unsichtbaren einen relevanten Schutz vor Entfernung liefern können. Somit lag der Fokus dieses Rechercheteils auf den unsichtbaren Algorithmen.

Die Recherche wurde mit den folgenden Suchbegriffen durchgeführt: *"watermarking survey"*, *"invisible watermarking"*, *"watermarking attack"*.

Dies führte bereits zu einer Vielzahl an aktuellen wissenschaftlichen Artikeln zu diesem Thema. Der Fokus lag dabei auf Publikationen, die einen Überblick über die existierenden Algorithmen verschaffen oder solche Algorithmen attackieren, da für diese im Allgemeinen auf bekannte, öffentlich zugängliche Tools zurückgegriffen wird. Dabei zeigte sich, dass wie in [theoretischen Grundlagen zu Wasserzeichen](#) beschrieben,

Im Abschnitt [Übersicht zu Wasserzeichen-Software](#) werden die in den Publikationen [27, 28, 29] verwendeten Implementierungen kurz vorgestellt und in diese Arbeit eingeordnet.

3.4. Literaturrecherche zu Forschungsdaten für Bild-Steganografie

Für die Arbeit mit den mobilen Steganografie-Anwendungen haben Newman et al. eine Bild-Datenbank aufgebaut, die öffentlich über eine Webseite zugänglich ist. Zunächst kann ausgewählt werden, ob Stego- oder nur Cover-Bilder selektiert werden sollen. Danach ermöglicht ein Eingabeformular das weitere Konfigurieren des gewünschten Datensatzes.

Für den Android-Bereich stehen fünf und für den iOS-Bereich sechs verschiedene Modelle zur Verfügung, auf denen Bilder aufgenommen wurden, während es für Android fünf und für iOS lediglich eine Steganografie-App zur Auswahl gibt. Des Weiteren können zwischen drei Einbettungsraten gewählt werden:

- Zwischen 0% und 10%
- Zwischen 10% und 20%
- Zwischen 20% und 40%

Schließlich kann die Belichtung auf *automatisch* und/oder *manuell* im Bereich von 10 bis 7000 ISO und einer Belichtungszeit zwischen $\frac{1}{11000}$ und $\frac{1}{2}$ gestellt werden.

Zwar wird auf der Webseite nicht die Anzahl der Bilder nach der gewünschten Einstellung benannt, Experimente mit den heruntergeladenen Daten zeigten jedoch, dass für die meisten Konfigurationskombinationen mehrere tausend Bilder zur Verfügung stehen, was der statistischen Auswertbarkeit der Daten Genüge tut. Zudem besticht die Verwendung der StegoAppDB damit, dass für die Steganografie-Apps keine Cover-Stego-Bildpaare mehr generiert werden müssen und so der Prototyp direkt auf diesen Daten angewendet werden kann.

Die konkret verwendete Konfiguration zum Herunterladen der StegoAppDB-Datensatzes für diese Arbeit wird im Abschnitt [Verwendeter Bild-Datensatz](#) beschrieben.

3.5. Implementierung eines Prototyps zur Merkmalsfindung steganografischer Bilddaten

Zur Umsetzung eines prototypischen Detektors für die Identifikation bestimmter Steganografie-Werkzeuge wurde in grundsätzlich zwei Schritten vorgegangen.

1. Bestimmung möglicher Merkmalskategorien
2. Recherche und Nutzung von Dateianalyse- und Steganalyse-Tools zur Findung der Merkmale, die auf die Verwendung bestimmter Stego-Tools hinweisen
3. Entwicklung des Detektors basierend auf gefundenen Merkmalen

3.5.1. Merkmalskategorien

Aus den vorgestellten [Verfahren zur Einbettung](#) lassen sich Merkmalskategorien ableiten, den den Domänen der Einbettungsverfahren entsprechen.

- Merkmale der Raum-Domäne: Dabei werden die Pixel der Bilder nach Auffälligkeiten untersucht. Dazu gehören statistischen Analysen von Pixel-Änderungen wie auch Analysen der LSBs der Pixel.
- Merkmale der Frequenz-Domäne: Entsprechend der Raum-Domäne werden statistischen Analysen oder LSB-Analysen auf den Koeffizienten von komprimierten Bildern durchgeführt.
- Merkmale der Struktur: Es werden Dateigrößen und weitere Metadaten der Cover- und Stego-Bilder verglichen.

3.5.2. Analyse-Tools

Zur Recherche wurde hauptsächlich die Google-Suchmaschine verwendet, da die meisten Analyse-Tools nicht im akademischen Umfeld entwickelt werden. Die nachfolgenden Tools wurden als potenziell nützlich identifiziert und aktiv verwendet.

Table 1. Verwendete Analyse-Tools

Name	Domäne(n)	Benutzeroberfläche	Kommentar
exiftool	Struktur	Kommandozeile	Gibt alle gefundenen Metadaten zu Mediendateien aus. Kann bei JPEGs auch grundlegende Informationen zu Koeffizienten liefern.
sherloq	Struktur, Raum	Grafisch	Ermöglicht Darstellung von Metadaten (mittels <i>exiftool</i> im Hintergrund). Kann verschiedene Filter auf Bilder legen und weiteres.
aletheia	Struktur, Raum, Frequenz	Kommandozeile	Bietet Ausgabe von Metadaten, statistische Analysen in der Raum-Domäne, Simulationen von Stego-Tools, Angriffe auf Steganografie mittels maschinellem Lernen



Weitere aus dem Linux-Umfeld sind zum Beispiel *binwalk* und *foremost*, die jedoch in dieser Arbeit nicht verwendet wurden.

Sämtliche Analysen wurden auf einem Macbook Pro M1 durchgeführt.

Insbesondere das Analyse-Tool *aletheia* stellte sich durch die Vielzahl an Funktionen als sehr nützlich heraus, da es in Python geschrieben, quelloffen und daher denkbar einfach zu erweitern oder in Python-Skripte oder Jupyter Notebooks als Bibliothek einzubinden ist.

Die in dieser Arbeit verwendete Version von *aletheia* wurde vom Autor angepasst, um den aktuellen Stand von Python nutzen zu können sowie die Erweiterung der Kommandozeilen-Befehle des Tools zu vereinfachen.

3.5.3. Vorstellung der Analysen

Es wurden drei Analysen aufgrund des [verwendeten Datensatzes](#) ausgewählt. Die Auswahl wurde so vorgenommen, da die meisten Stego-Apps LSB-Verfahren implementieren, weshalb Änderungen im Stego-Bild in der *Raum-Domäne* **höchstwahrscheinlich**, in der *Struktur* **wahrscheinlich** und in der *Frequenz-Domäne* **unwahrscheinlich** zu finden sind.

Table 2. Ausgewählte Analysen

Name	Domäne	Blind? (Genügt Stego-Bild)	Ziele
Metadatenvergleich	Struktur	Nein	* Herausarbeiten von Veränderungen der Metadaten im Stego-Bild im Vergleich zum Cover-Bild. * Vergleich nicht nur innerhalb der Paare, sondern auch über alle Paare mit der gleichen eingebetteten Nachricht hinweg
Dateigrößenunterschied	Struktur	Nein	* Analyse der Veränderung der Dateigröße zwischen Cover- und Stego-Bild * Herausarbeiten möglicher Zusammenhänge oder identifizierbarer Intervalle in der prozentualen Veränderung
LSB-Extraktion	Raum	* Bei <i>Analyse</i> : Nein , da nach der eingebetteten Nachricht gesucht wird, die bekannt sein muss. * Bei <i>Detektion</i> : Ja , da nur im Stego-Bild nach Signaturen, etc. gesucht wird.	* Identifikation der verwendeten Parameter zur LSB-Einbettung. * Identifikation möglicher eingebetteter Signaturen, Längenangaben der Nachricht, etc.

Zu jeder der in der Tabelle vorgestellten Analysen wurden die folgenden algorithmischen Vorgehensweisen gewählt.

Metadaten-Vergleich

Pro Stego-App:

1. Lese Metadaten von erstem Cover- und Stego-Bild-Paar als Zuordnungstabellen c und s aus.
2. Vergleiche c und s auf Unterschiede und schreibe diese in neue Zuordnungstabelle u .
3. Für jedes weitere Paar: Wiederhole 1. und 2. und kombiniere u mit vorherigem u mittels Schnittmenge.

Implementierung

```
def metadata(filename) -> dict:
    """Return the metadata of an image."""
    image = Image.open(filename)
    jfif_data = _get_jfif_data(image)
    exif_data = _get_exif_data(image)
    return {
        'Size': os.path.getsize(filename),
        'Format': image.format,
        'Mode': image.mode,
        'Height': image.height,
        'Width': image.width,
        **jfif_data,
        **exif_data
    }

def metadata_diff(cover, stego) -> dict:
    """Return the differences between the metadata of two images."""
    cover_data = metadata(cover)
    stego_data = metadata(stego)
    diff = {}
    for key in cover_data.keys() | stego_data.keys():
        if cover_data.get(key) != stego_data.get(key):
            diff[key] = (cover_data.get(key), stego_data.get(key))

    return diff
```

Schritt 1 ist in `metadata()` und Schritt 2 in `metadata_diff()` implementiert. Schritt 3 wurde in der Python-Konsole bzw. Jupyter Notebook umgesetzt.

Dateigrößenunterschied

Für jedes Cover-Stego-Paar pro Stego-App:

1. Lese Dateigröße von Cover als c und von Stego als s aus.
2. Berechne Differenz $d = s - c$ und prozentualer Unterschied $p = \frac{d}{c} \cdot 100$
3. Erstelle gemeinsame Übersicht (bspw. als Grafik) aller prozentualen Unterschiede.

Implementierung

```
def size_diff(image_pairs, *, payload_length=None, verbose=False):
    """Calculate the size difference between cover and stego images.

    The image_pairs is a list of tuples with the cover and stego image paths.

    :param image_pairs: list of tuples with cover and stego image paths
```



```

:param payload_length: payload size in bytes
:param verbose: whether to print the differences
:return: the average size difference
"""

results = []
for cover, stego in image_pairs:
    cover_size = os.path.getsize(cover)
    stego_size = os.path.getsize(stego)
    diff = stego_size - cover_size
    if payload_length:
        diff -= payload_length
    percent = diff / cover_size * 100
    if verbose:
        click.echo(f"Cover: {cover_size} bytes, Stego: {stego_size} bytes,
Diff: {diff} bytes ({percent:.2f}%)")
    results.append((cover, stego, diff, percent))
return results

```

Schritt 1 und 2 ist in `size_diff()` implementiert. Schritt 3 wurde im Jupyter Notebook umgesetzt, in die `results`-Liste zur Visualisierung genutzt wurde.

LSB-Extraktion

Für jedes Cover-Stego-Paar pro Stego-App:

1. Extrahiere die Least n Significant Bits aus den Farbkanälen R , G , B oder A pro Pixel pro Stego-Bild aus und füge diese im Big-Endian- oder Little-Endian-Format aneinander. Daraus ergeben sich die Parameter *Anzahl der LSBs* n , *verwendete Farbkanäle* und das *Endian-Format*.
2. Suche nach der eingebetteten Nachricht.
3. Suche nach wiederkehrenden Mustern in allen extrahierten Nachrichten ohne Zusammenhang mit der Nachricht (*Signatur*) oder im mit Zusammenhang mit der Nachricht (bspw. *Nachrichtenlänge* in den ersten 4 Bytes).
4. Vergleiche Ergebnisse aller Extraktionen zur Validierung.

Implementierung

```

def _extract_bits_opt_little(data):
    div = 8 // bits
    message = np.zeros(len(data) // div, dtype=np.uint8)
    mask = (1 << bits) - 1
    for i in range(div):
        shift = bits * i
        message |= (data[i::div] & mask) << shift
    return message

```

```

def _extract_bits_opt_big(data):
    div = 8 // bits
    message = np.zeros(len(data) // div, dtype=np.uint8)
    mask = (1 << bits) - 1
    for i in range(div):
        shift = 8 - bits - (bits * i)
        message |= (data[i::div] & mask) << shift
    return message

def _extract_bits_little(data):
    msg_byte = 0
    shift = 0
    message = []
    mask = (1 << bits) - 1
    for byte in data:
        msg_byte |= (byte & mask) << shift
        shift += bits
        if shift >= 8:
            tmp = msg_byte >> 8
            message.append(msg_byte & 0xFF)
            msg_byte = tmp
            shift -= 8
    return np.array(message, dtype=np.uint8)

def _extract_bits_big(data):
    msg_byte = 0
    shift = 8 - bits
    message = []
    mask = (1 << bits) - 1
    for byte in data:
        msg_byte |= (byte & mask) << shift
        shift += bits
        if shift <= 0:
            tmp = msg_byte >> 8
            message.append(msg_byte & 0xFF)
            msg_byte = tmp
            shift += 8
    return np.array(message, dtype=np.uint8)

```

Schritt 1 ist mittels der `extract_bits*()`-Funktionen umgesetzt. Es existiert eine **optimierte** Version jeweils für das Little- und Big-Endian-Format, wenn die Least n Significant Bits 1 oder ein Vielfaches von 2 sind. Ansonsten werden nicht-optimierten Versionen zur Extraktion verwendet. `np.zeros()` sowie `np.array()` stammt aus der `numpy`-Bibliothek.

Weitere potenziell relevante Analysen für den Datensatz wie die R/S-Analyse wurden aus Gründen des Umfangs dieser Arbeit nicht durchgeführt. Die dazu nötigen Algorithmen sind ebenfalls im Analyse-Tool *aletheia* implementiert und sollten für eine nächste Ausbaustufe des Detektors in Betracht gezogen werden.

3.5.4. Implementierung eines prototypischen Detektors

Die konkreten Ziele des prototypischen Detektors sind:

1. Bestimmung des zur Einbettung verwendeten Tools basierend auf Cover- und Stego-Bild
2. Nachvollziehbarkeit des Bestimmungsprozesses
3. Einfacher modularer Aufbau zur einfachen Erweiterbarkeit
4. Deterministische Ergebnisse (vorerst Verzicht auf maschinelles Lernen zur Erkennung von Merkmalen)

Basierend auf diesen Zielen wurden mögliche bestehende Lösungen untersucht. Neben den vielen proprietären Malware-Scanner fiel die quelloffene Lösung YARA des Malware-Analyse-Portals VirusTotal als möglicher Detektor für diese Arbeit auf.

YARA ermöglicht die Analyse von Dateien (mit dem Fokus auf Malware) anhand selbst definierter Regeln. Diese Regeln werden in einer domänen spezifischen Sprache geschrieben und an YARA gemeinsam mit der zu analysierenden Datei übergeben. Die Webseite des Tools zeigt das folgende Beispiel einer Regel:

Beispiel-Regel von der YARA-Webseite

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

Die Regel heißt *silent_banker*, weist unter anderem eine Beschreibung und eine Gefahrenniveau *threat_level* sowie mehrere Strings und eine Bedingung auf. In der Sektion *strings* werden die Strings Variablen zugeordnet, welche in der Bedingung eine nach der anderen mit der Eingabedatei verglichen werden, bis eine Übereinstimmung gefunden wurde. Mittels Modulen erlaubt YARA den Zugriff auf verschiedene weitere Daten der Eingabedatei, sodass auch dynamische Werte untersucht werden können.

Damit YARA jedoch nutzbar zur Umsetzung der zuvor vorgestellten Analysen ist, muss insbesondere das erste Ziel möglich sein. YARA erlaubt jedoch nur eine Eingabedatei, wodurch alle nicht-blinden Analysen nicht mehr direkt möglich sind. Wäre diese Bedingung erfüllt, ließe sich zumindest die Dateigrößenanalyse als Regel definieren. Damit auch der Metadatenvergleich und

die LSB-Extraktion möglich werden, müsste YARA die entsprechenden Daten bereitstellen, sodass von Regeln daraufzugegriffen werden kann. Diese Funktion ist über selbst entwickelte Module in der Programmiersprache C zwar möglich, wurde aber aufgrund der Komplexität der Sprache selbst sowie des Build-Systems an dieser Stelle nicht weiterverfolgt.

Dennoch diente YARA als Inspiration eines selbst-entwickelten Tools in Python, dass auch direkt auf *alethia* zur Detektion zurückgreift. Wie in YARA werden Regeln definiert, die auf die Eingabe-Bildpaare angewendet werden. Die Regeln sind im YAML-Format in einer Konfigurationsdatei definiert, sodass kein eigener Parser wie bei einer domänenspezifischen Sprache entwickelt werden musste.

Der Aufbau der Konfigurationsdatei ist wie folgt:

Konfigurationsdatei des Detektors

```
isd: "1" ①
tools: ②
  - name: PixelKnot ③
    tags: [ Android, F5 ] ④
rules: ⑤
  - name: ISA.PixelKnot.File-Size-Diff ⑥
    desc: Check if the file size difference is maximum -3%. ⑦
    tools: ⑧
      - name: PixelKnot ⑨
        weight: 5 ⑩
    match: ⑪
      value: attacks.size_diff([(cover.path, stego.path)])[0][3] ⑫
      cond: -3 <= value < 0 ⑬
```

- ① Versionsnummer (aktuell nur "1", "1.0" oder "1.0.0") und Marker, dass diese Konfigurationsdatei für den Detektor bestimmt ist
- ② Tools, die durch die nachfolgend definierten Regeln detektiert werden können
- ③ Name eines Stego-Tools
- ④ Frei wählbare Tags zur Zuordnung des Tools zu bestimmten Kategorien
- ⑤ Einstieg zum von oben nach unten abzuarbeitenden Regelbaum
- ⑥ Name der ersten Regel
- ⑦ Beschreibung der ersten Regel
- ⑧ Detektierbare Tools durch diese Regel
- ⑨ Name des detektierbaren Tools
- ⑩ Gewichtung, falls die Regel erfolgreich war
- ⑪ Definition der Zuordnung
- ⑫ Der zu vergleichende Wert
- ⑬ Eine Bedingung, die wahr (**True**) oder falsch (**False**) zurückgeben muss

In jeder Regel kann der **match** auch direkt ein Python-Ausdruck sein. Ansonsten können wie im

Beispiel `value` und `cond` verwendet werden, um den zu vergleichenden Wert ausgeben zu lassen. Sowohl `value` als auch `cond` sind Python-Ausdrücke.

Im Beispiel wird die Dateigröße von Cover- und Stego-Bild ausgewertet und aus dem ersten Element der Ergebnisliste (`[0]`) das vierte Element (`[3]`) ausgelesen, welches den prozentualen Unterschied beinhaltet. Wenn der Unterschied zwischen -3 % und 0 % liegt, könnte PixelKnot für die Einbettung verwendet worden sein.

Chapter 4. Ergebnisse

Die folgenden Ergebnisse sind jeweils zu einem Abschnitt in [Methodik](#) zugeordnet.

4.1. Malware-Vorkommnisse im Zusammenhang mit Steganografie

In der folgenden Tabelle sind die ersten fünf Einträge der zuvor gesammelten Malware-Vorkommnisse zu sehen.

Table 3. Auszug aus Malware-Daten

Name	Description	Type	Platforms	Version	Created	Last Modified	MITRE ID	Techniques Used	Contributors	Associated Software	References
Zox	Zox is a remote access tool that has been used by Axiom since at least 2008.[1]	MALWARE	Windows	1.0	2022-01-09	2023-03-20	S0672	Zox has used the .PNG file format for C2 communications.[1]		Gresim, ZoxRPC, ZoxPNG	

Name	Description	Type	Platforms	Version	Created	Last Modified	MITRE ID	Techniques Used	Contributors	Associated Software	References
Diavol	Diavol is a ransomware variant first observed in June 2021 that is capable of prioritizing file types to encrypt based on a pre-configured list of extensions defined by the attacker. Diavol has been deployed by Bazar and is thought to have potential ties to Wizard	MALWARE	Windows	1.0	2021-11-12	2022-04-15	S0659	Diavol has obfuscated its main code routines within bitmap images as part of its anti-analysis techniques.[1]	Massimiliano Romano, BT Security		
	Wizard										

[illegible]

Name	Description	Type	Platforms	Version	Created	Last Modified	MITRE ID	Techniques Used	Contributors	Associated Software	References
ObliqueRAT	ObliqueRAT is a remote access trojan, similar to Crimson, that has been in use by Transparent Tribe since at least 2020.[1][2]	MALWARE	Windows	1.0	2021-09-08	2021-10-15	S0644	ObliqueRAT can hide its payload in BMP images hosted on compromised websites.[1]			

Name	Description	Type	Platforms	Version	Created	Last Modified	MITRE ID	Techniques Used	Contributors	Associated Software	References
Raindrop	Raindrop is a loader used by APT29 that was discovered on some victim machines during investigations related to the Solar Winds Compromise. It was discovered in January 2021 and was likely used since at least May 2020.[1][2]	MALWARE	Windows	1.2	2021-01-19	2023-03-27	S0565	Raindrop used steganography to locate the start of its encoded payload within legitimate 7-Zip code.[1]			

Der Ausschnitt aus der vollständigen Tabellen ([siehe Anhang](#)) wurden mit den folgenden Schritten erstellt:

Erstellung des Tabellenauszugs

1. Gesamttabelle nach Werten in der Spalte **Created** absteigend sortieren
2. Zeilen, deren **Name** dem regulären Ausdruck `[\\"{}@]` entsprechen, entfernen
3. Spalte **Page Content** entfernen
4. Ersten 5 Ergebnisse auswählen

Der Auszug zeigt fünf Malware-Vorkommnisse, die erstmals im Zeitraum von Juni 2021 bis Januar 2022 registriert wurden. Die Malware *Zox* und *Raindrop* erfuhren Aktualisierungen im März 2023 und waren zumindest bis zu diesem Zeitpunkt noch aktiv. Jede der gezeigten Malwares wurde für die Ausführung auf dem Betriebssystem Windows entwickelt. Drei Malwares nutzen Bitmap-Bilder, wobei davon eines auch auf JPEG-Bilder zurückgreift, während nur *Zox* PNG-Bilder verwendet. Der gezeigte Auszug ist dabei jedoch voll repräsentativ, wie die nächsten Auswertungen zeigen.

Table 4. Anzahl der Malwares pro Steganographie-Art

Steganographie-Art	Anzahl der Malwares
LSB	4
XOR	2
End of File	2
Unknown	29

TODO:

- Mit aktuellen Daten füllen!
- Mit Vorkommen pro Jahr angeben, um Shift zu LSB zu zeigen

Table 5. Anzahl der Malwares pro Plattform

Plattform	Anzahl der Malwares
Windows	27
Android	1
macOS	1
Linux	2
Unbekannt	16

TODO:

Mit aktuellen Daten füllen!

Table 6. Anzahl der Malwares pro Cover-Typ

Cover-Typ	Anzahl der Malwares
JPG	5
PNG	10
BMP	4
GIF	0

Cover-Typ	Anzahl der Malwares
TIFF	0
PDF	2
Unbekannt	24

TODO:

Mit aktuellen Daten füllen!

4.2. Steganografie- und Wasserzeichen-Software

Die anschließenden Unterabschnitte listen die in der Recherche gefundenen Softwares auf und vergleicht diese nach ihren Funktionen in einer Gesamtübersichtstabelle miteinander.

4.2.1. Übersicht zu Steganografie-Software

In den folgenden Tabellen werden die Steganografie-Programme vorgestellt. Die erste Tabelle zeigt die Tools, die auf Desktop-Plattformen wie Windows, macOS und Linux laufen, während die zweite die Apps auf den mobilen Plattformen Android und iOS präsentiert. Beide Tabellen folgen dem gleichen Schema. Zunächst wird die Software, dann ihre unterstützten Plattformen, ihrer Steganografie-Domäne (siehe Theorie), ihre unterstützten Steganografie-Algorithmen und schließlich zugehörige Referenzen benannt.

Desktop-Tools

1. jsteg [23, 22]
2. outguess [23, 22]
3. steghide [23, 22]
4. stegify
5. f5 [23, 22]
6. stegpy [23]
7. lsbsteg
8. stegolsb

Table 7. Desktop-Anwendungen

Name	Plattformen	Domäne	Algorithmen	Referenzen
jsteg	Windows, macOS, Linux	Raum	LSB	[23, 22]
steghide	Windows, macOS, Linux	Raum	LSB	[23, 22]
stegpy	Windows, macOS, Linux	Raum	LSB	[23]

Name	Plattformen	Domäne	Algorithmen	Referenzen
outguess	Windows, macOS, Linux	Frequenz	Outguess	[23, 22]
f5	Windows, macOS, Linux	Frequenz	F5	[23, 22]

Mobile-Apps

1. PixelKnot
2. Passlok
3. MobiStego
4. PocketStego
5. Steganography-Meznik
6. Pictograph

Table 8. Mobile Anwendungen

Name	Plattformen	Domäne	Algorithmen	Referenzen
PixelKnot	Android	Frequenz	F5	[26]
Passlok Privacy	Android	Frequenz		[26]
MobiStego	Android	Raum	LSB	[26]
Steganography-Meznik	Android	Raum	LSB	[26]
Pictograph	iOS	Raum	LSB	[26]

4.2.2. Übersicht zu Wasserzeichen-Software

Der aktuelle Stand der Forschung zeigt, dass insbesondere die Verwendung von neuronalen Netzen zur Einbettung von Wasserzeichen in Bildern den aktuell höchsten Schutz vor Angriffen in diesem Fachbereich bieten. Da unsichtbare Wasserzeichen im Gegensatz zu Steganografie-Tools oder auch -Malware einen Anwendungsbereich im Umfeld von Medienkonzernen finden und weniger von Endbenutzern verwendet werden, lassen sich vor allem komplexe wissenschaftlich belegte Implementierungen finden.

1. [DwtDcdSvd](#) [30]
2. [StegaStamp](#) [31]
3. [SSL Watermarking](#) [32]
4. [Stable Signature](#) [33]
5. [Tree-Ring Watermarks](#) [34]
6. [invisible-watermark](#)
7. [Easy Watermark](#)

Name	Plattformen	Domäne	Algorithmen	Referenzen
DwtDcdSvd	Windows, macOS, Linux	Frequenz		[30]
StegaStamp	Windows, macOS, Linux	Frequenz	Neuronales Netzwerk	[31]
SSL Watermarking	Windows, macOS, Linux		Neuronales Netzwerk	[32]
Stable Signature	Windows, macOS, Linux		Neuronales Netzwerk	[33]
Tree-Ring Watermarks	Windows, macOS, Linux		Neuronales Netzwerk	[34]
Tree-Ring Watermarks	Android		Neuronales Netzwerk	Easy Watermark

4.2.3. Gesamtübersicht

TODO:

- Diagramm der Gesamtübersicht mit:
 - Domäne
 - Kategorie (Desktop, App, WZ)

4.3. Verwendeter Bild-Datensatz

Der ausgewählte Datensatz beinhaltet Cover-Stego-Bildpaare der zuvor beschriebenen mobilen Stego-Apps. Diese setzen hauptsächlich Steganografie in der Raumdomäne ein, wobei PixelKnot in der Frequenz-Domäne arbeitet. Um den Einsatz von Information Hiding durch Malware besser widerzuspiegeln, wurde aus den Cover-Bildern des Datensatzes zudem Stego-Bilder mit den Verfahren der Struktur-Einbettung erstellt.

Die Einstellung des Suchformulars der StegoAppDB wurde wie folgt vorgenommen:

Search For: ☒ Stego Images ☐ Original Images

Stego-related Images:

☒ Stego images

☒ Include pre-stego images (cover and input) ?

☐ Include original images ?

Android

Embedding Program:

- ☒ PixelKnot (JPG)
- ☒ Passlok (JPG)
- ☒ MobiStego (PNG)
- ☒ PocketStego (PNG)
- ☒ Steganography-Meznik (PNG)

Apple

☒ Pictograph (PNG)

Original Image Source Device:

	Device Number			
	1	2	3	4
OnePlus 5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pixel 1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pixel 2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Samsung Galaxy S7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Samsung Galaxy S8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

	Device Number			
	1	2	3	4
iPhone6s	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone6sPlus	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone7Plus	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhoneX	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Embedding Rate:

☒ 0% < rate ≤ 10%
 ☒ 10% < rate ≤ 20%
 ☒ 20% < rate ≤ 40%

Original Image Exposure Settings:

☒ Auto Exposure

☐ Manual Exposure

ISO

10 - 7000

Exposure Time

1/11000 - 1/2

Number of Images:

Estimated Download Size:

24468

18.53 Gigabyte(s)

Download
Search Again

Abbildung 5. Einstellung StegoAppDB

Es wurden lediglich immer das erste Gerät pro Gerätetyp ausgewählt und die manuelle Belichtungsdauer nicht selektiert, da sonst die Größe des Datensatzes um nahezu das Zehnfache angestiegen wäre. (siehe [Abbildung 6](#))

4.4. Übersicht von Merkmalen steganografischer Bilddaten

- Vorstellung Metadaten
- Vorstellung Signaturen in LSB-Ebenen
- Vorstellung Dateigrößendifferenzen

4.5. Auswertung des Detektors

- Vorstellung Ergebnisse der Detektionen

Chapter 5. Zusammenfassung

Chapter 6. Ausblick

Kamera-Fingerprinting

Bibliografie

- [1] W. Gemoll, K. Vretska, T. Aigner, und R. Wachter, *Griechisch-deutsches Schul- und Handwörterbuch*, 10., Völlig neu bearb. Aufl., [Nachdr.]. Oldenbourg.
- [2] S. Singh, *The code book: the science of secrecy from ancient Egypt to quantum cryptography*, 1. Ed. Anchor Books, 2000.
- [3] R. Benson, „Data Infiltration - DFIQ (Digital Forensics Investigative Questions)“. Zugegriffen: Feb. 09, 2024. [Online]. Verfügbar unter: <https://dfiq.org/scenarios/S1002/>.
- [4] R. Benson, „Data Exfiltration - DFIQ (Digital Forensics Investigative Questions)“. Zugegriffen: Feb. 09, 2024. [Online]. Verfügbar unter: <https://dfiq.org/scenarios/S1001/>.
- [5] J. Dittmann, „Motivation und Einführung“, in *Digitale Wasserzeichen: Grundlagen, Verfahren, Anwendungsgebiete*, J. Dittmann, Hrsg. Springer, 2000, S. 1–7.
- [6] F. A. P. Petitcolas, R. J. Anderson, und M. G. Kuhn, „Information hiding-a survey“, *Proceedings of the IEEE*, Bd. 87, Nr. 7, S. 1062–1078, Juli 1999, doi: 10.1109/5.771065.
- [7] T. Agrawal, „A Survey On Information Hiding Technique Digital Watermarking“, Juni 2015, Bd. 3, doi: 10.18479/ijeedc/2015/v3i8/48358.
- [8] E. Hamilton, „JPEG File Interchange Format specification v1.02“. C-Cube Microsystems, Sep. 01, 1992, [Online]. Verfügbar unter: <http://www.w3.org/Graphics/JPEG/jfif3.pdf>.
- [9] M. Begum und M. S. Uddin, „Digital Image Watermarking Techniques: A Review“, *Information*, Bd. 11, Nr. 2, S. 110, Feb. 2020, doi: 10.3390/info11020110.
- [10] H. Lin, „Attribution of Malicious Cyber Incidents: From Soup to Nuts“, *Journal of International Affairs*, Bd. 70, Nr. 1, S. 75–137, 2016, Zugegriffen: März 09, 2024. [Online]. Verfügbar unter: <https://www.jstor.org/stable/90012598>.
- [11] L. Caviglione und W. Mazurczyk, „Never Mind the Malware, Here’s the Stegomalware“, *IEEE Security & Privacy*, Bd. 20, Nr. 5, S. 101–106, Sep. 2022, doi: 10.1109/MSEC.2022.3178205.
- [12] L, „lucacav/steg-in-the-wild“. Feb. 01, 2024, Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://github.com/lucacav/steg-in-the-wild>.
- [13] R. Chaganti, V. Ravi, M. Alazab, und T. D. Pham, „Stegomalware: A Systematic Survey of MalwareHiding and Detection in Images, Machine LearningModels and Research Challenges“, Nr. arXiv:2110.02504. arXiv, Okt. 06, 2021, Zugegriffen: Dez. 07, 2023. [Online]. Verfügbar unter: <http://arxiv.org/abs/2110.02504>.
- [14] cybleinc, „Stegomalware - Identifying possible attack vectors“. Aug. 04, 2022, Zugegriffen: Dez. 07, 2023. [Online]. Verfügbar unter: <https://cyble.com/blog/stegomalware-identifying-possible-attack-vectors/>.
- [15] „Alibaba OSS Buckets Compromised to Distribute Malicious Shell Scripts via Steganography“. Juli 21, 2022, Zugegriffen: Dez. 07, 2023. [Online]. Verfügbar unter: <https://www.trendmicro.com/>

en_us/research/22/g/alibaba-oss-buckets-compromised-to-distribute-malicious-shell-sc.html.

- [16] „MITRE ATT&CK®“. Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/>.
- [17] „Malpedia (Fraunhofer FKIE)“. Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://malpedia.caad.fkie.fraunhofer.de/>.
- [18] „Software \textbar MITRE ATT&CK®“. Zugegriffen: Feb. 23, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/software/>.
- [19] „Sliver, Software S0633 \textbar MITRE ATT&CK®“. Zugegriffen: Feb. 23, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/software/S0633/>.
- [20] U. Pilia, R. Tanwar, P. Gupta, und T. Choudhury, „A roadmap of steganography tools: conventional to modern“, *Spatial Information Research*, Bd. 29, Nr. 5, S. 761–774, Okt. 2021, doi: 10.1007/s41324-021-00393-7.
- [21] U. Pilia, R. Tanwar, und K. Kaushik, „Steganography Tools and Their Analysis Concerning Distortion in Stego Image“, in *Advances in Data Science and Computing Technologies*, 2023, S. 531–538, doi: 10.1007/978-981-99-3656-4_54.
- [22] V. Verma, S. K. Muttoo, und V. B. Singh, „Detecting Stegomalware: Malicious Image Steganography and Its Intrusion in Windows“, in *Security, Privacy and Data Analytics*, 2022, S. 103–116, doi: 10.1007/978-981-16-9089-1_9.
- [23] „DominicBreuker/stego-toolkit: Collection of steganography tools - helps with CTF challenges“. Zugegriffen: Feb. 29, 2024. [Online]. Verfügbar unter: <https://github.com/DominicBreuker/stego-toolkit>.
- [24] W. Chen, Y. Wang, Y. Guan, J. Newman, L. Lin, und S. Reinders, „Forensic Analysis of Android Steganography Apps“, in *Advances in Digital Forensics XIV*, 2018, S. 293–312, doi: 10.1007/978-3-319-99277-8_16.
- [25] W. Chen, L. Lin, M. Wu, und J. Newman, „Tackling Android Stego Apps in the Wild“, Nr. arXiv:1808.00430. arXiv, Aug. 01, 2018, doi: 10.48550/arXiv.1808.00430.
- [26] J. Newman u. a., „StegoAppDB: a Steganography Apps Forensics Image Database“, Nr. arXiv:1904.09360. arXiv, Apr. 19, 2019, doi: 10.48550/arXiv.1904.09360.
- [27] B. An u. a., „Benchmarking the Robustness of Image Watermarks“, Nr. arXiv:2401.08573. arXiv, Jan. 22, 2024, doi: 10.48550/arXiv.2401.08573.
- [28] H. Mareen, L. Antchougov, G. Van Wallendael, und P. Lambert, „Blind Deep-Learning-Based Image Watermarking Robust Against Geometric Transformations“, Nr. arXiv:2402.09062. arXiv, Feb. 14, 2024, Zugegriffen: Feb. 29, 2024. [Online]. Verfügbar unter: <http://arxiv.org/abs/2402.09062>.
- [29] X. Zhao u. a., „Invisible Image Watermarks Are Provably Removable Using Generative AI“, Nr. arXiv:2306.01953. arXiv, Aug. 06, 2023, doi: 10.48550/arXiv.2306.01953.
- [30] K. A. Navas, M. C. Ajay, M. Lekshmi, T. S. Archana, und M. Sasikumar, „DWT-DCT-SVD based

watermarking“, in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, Jan. 2008, S. 271–274, doi: 10.1109/COMSWA.2008.4554423.

[31] M. Tancik, B. Mildenhall, und R. Ng, „StegaStamp: Invisible Hyperlinks in Physical Photographs“, Nr. arXiv:1904.05343. arXiv, März 25, 2020, doi: 10.48550/arXiv.1904.05343.

[32] P. Fernandez, A. Sablayrolles, T. Furon, H. Jégou, und M. Douze, „Watermarking Images in Self-Supervised Latent Spaces“, Nr. arXiv:2112.09581. arXiv, März 23, 2022, doi: 10.48550/arXiv.2112.09581.

[33] P. Fernandez, G. Couairon, H. Jégou, M. Douze, und T. Furon, „The Stable Signature: Rooting Watermarks in Latent Diffusion Models“, Nr. arXiv:2303.15435. arXiv, Juli 26, 2023, doi: 10.48550/arXiv.2303.15435.

[34] Y. Wen, J. Kirchenbauer, J. Geiping, und T. Goldstein, „Tree-Ring Watermarks: Fingerprints for Diffusion Images that are Invisible and Robust“, Nr. arXiv:2305.20030. arXiv, Juli 03, 2023, doi: 10.48550/arXiv.2305.20030.

Anhang A: Tabellen

`#[format=csv,separator=;,options="header,breakable",caption="Umfängliche Malware-Daten"]`
`#|=== #include:../notebooks/data/malware-full-data.csv[] #|===`

Anhang B: Abbildungen

figures::[]

Anhang C: Quellcode

Anhang D: Sonstiges

Search For: ☒ Stego Images ☐ Original Images

Stego-related Images:

☒ Stego images
☒ Include pre-stego images (cover and input) [?](#)
☐ Include original images [?](#)

Embedding Program:

Android
☒ PixelKnot (JPG)
☒ Passlok (JPG)
☒ MobiStego (PNG)
☒ PocketStego (PNG)
☒ Steganography-Meznik (PNG)

Apple
☒ Pictograph (PNG)

Original Image Source Device:

	Device Number			
	1	2	3	4
OnePlus 5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pixel 1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pixel 2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Samsung Galaxy S7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Samsung Galaxy S8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

	Device Number			
	1	2	3	4
iPhone6s	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone6sPlus	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone7Plus	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhone8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
iPhoneX	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Embedding Rate:

☒ 0% < rate ≤ 10% ☒ 10% < rate ≤ 20% ☒ 20% < rate ≤ 40%

Original Image Exposure Settings:

ISO

Exposure Time

☒ Auto Exposure
☒ Manual Exposure

10 - 7000

1/11000 - 1/2

10 - 7000

1/11000 - 1/2

Number of Images:

244680

Estimated Download Size:

175.15 Gigabyte(s)

Download

Search Again

Abbildung 6. Einstellung StegoAppDB nicht verwendet