

Fingerprinting-Merkmale für Bilddaten unter Einfluss von Information Hiding

Masterarbeit
zur Erlangung des Grades **Master of Science (M.Sc.)**
des Fachbereichs Informatik und Medien der
Technischen Hochschule Brandenburg

vorgelegt von
Fabian W. E. Loewe

Betreuer Prof. Dr. Claus Vielhauer
Zweitgutachter Benedikt Michaelis, M.Sc.

Potsdam, 20. Mai 2024

Abstract

In this master thesis, an overview of the topic of information hiding within image files will be created. Techniques used by embedding tools and malware will be focused on. Research of current malware occasions will be conducted to show the relevancy of the topic and to provide a starting point for further investigations. Embedding tools that use algorithms comparable to those found in malware, and a dataset of cover stego image pairs will be selected. The image pairs will be examined with the help of analysis tools and the results will be used to extract fingerprinting features that provide hints about the usage of certain algorithms. Based on these findings, a detector prototype will be implemented and evaluated. Finally, the outlook will show further research areas related to information hiding based on the discussed results of this work.

Keywords

information hiding, image steganography, steganografic malware, information hiding detection

Zusammenfassung

In dieser Masterarbeit wird zunächst ein Überblick zum Information Hiding in Bilddateien verschafft. Der Fokus wird auf durch Einbettungswerkzeuge und Malware eingesetzte Information Hiding-Techniken gelegt. Eine Untersuchung aktueller Malware-Vorkommnisse wird durchgeführt, um die Relevanz der Thematik aufzuzeigen und einen Ausgangspunkt für weitere Untersuchungen zu erstellen. Anhand der Funde werden Tools, die algorithmisch verwandte Verfahren zu den untersuchten Malware-Vorkommnissen anwenden, und Forschungsdaten mit Cover- und Stego-Bild-Paaren ausgewählt. Die Bildpaare werden daraufhin mithilfe von Analyse-Werkzeugen untersucht und aus den Ergebnissen steganografische Merkmale extrahiert, welche zum Fingerprinting bestimmter Algorithmen genutzt werden können. Basierend auf diesen Merkmalen wird ein Detektor-Prototyp entwickelt, welcher mit den Forschungsdaten evaluiert wird. Schließlich werden im Ausblick weitere Forschungsrichtungen im Bereich des Information Hiding basierend auf den Ergebnissen dieser Arbeit vorgestellt.

Schlüsselwörter

Information Hiding, Bild-Steganografie, Steganografische Malware, Detektion von Information Hiding

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich, Fabian W. E. Loewe, die vorliegende Arbeit selbstständig verfasst habe, dass ich sie zuvor an keiner anderen Hochschule als Prüfungsleistung eingereicht habe und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht.

Ort, Datum

Unterschrift

Danksagungen

Das Thema dieser Arbeit ist aus dem Projekt ATTRIBUT heraus motiviert. Die Untersuchungen und Ergebnisse in dieser Arbeit wurden im Wintersemester 23/24 an der Technischen Hochschule Brandenburg erstellt und werden im Projekt weiterverwendet. Das Projekt ist durch die Agentur für Innovation in der Cybersicherheit GmbH: Forschung zu „Existenzbedrohenden Risiken aus dem Cyber- und Informationsraum – Hochsicherheit in sicherheitskritischen und verteidigungsrelevanten Szenarien“ (HSK) beauftragt. (siehe <https://www.cyberagentur.de/tag/hsk/>, auch in <https://attribut.cs.uni-magdeburg.de/>)

Ich möchte mich bei meinen Betreuern Prof. Dr. Claus Vielhauer und M.Sc. Benedikt Michaelis für ihre Unterstützung bedanken.

Ebenso bedanke ich mich bei der Kommilitonin, die sich im Vorfeld im Rahmen einer Projektarbeit auch mit dem Thema Fingerprinting im Bild-Bereich beschäftigt hat, und für mich sehr wertvolle Vorarbeit geleistet hat.

Inhaltsverzeichnis

Abstract	ii
Zusammenfassung	iii
Danksagungen	v
1. Einleitung	1
1.1. Motivation	1
1.2. Zielsetzung	2
1.3. Forschungsfragen und Hypothesen	3
1.4. Aufbau der Arbeit	3
2. Theoretische Grundlagen	4
2.1. Information Hiding	4
2.1.1. Anwendungsgebiete	4
2.1.2. Kriterien	5
2.1.3. Verfahren	5
2.2. Merkmalsanalyse	8
2.2.1. Metriken	8
3. Methodik	10
3.1. Suchmaschinen	10
3.2. Literaturrecherche zu Malware-Vorkommnissen im Zusammenhang mit Bild- Steganografie	11
3.2.1. Manuelle Recherche	11
3.2.2. Automatisierte Sammlung aus Datenbanken	12
3.3. Literaturrecherche zu Steganografie- und Wasserzeichen-Tools	21
3.3.1. Steganografie-Software	22
3.3.2. Wasserzeichen-Software	22
3.4. Literaturrecherche zu Forschungsdaten für Bild-Steganografie	23
3.5. Implementierung eines Prototyps zum Fingerprinting steganografischer Verfahren in Bilddaten	23
3.5.1. Merkmalskategorien	24
3.5.2. Analyse-Tools	24
3.5.3. Vorstellung der Analysen	25
3.5.4. Implementierung des Detektor-Prototyps	30
3.5.5. Auswertung des Detektors	35
4. Ergebnisse	39
4.1. Malware-Vorkommnisse im Zusammenhang mit Information Hiding	39
4.1.1. Zusammenfassung der Malware-Vorkommnisse	42
4.2. Steganografie- und Wasserzeichen-Software	43
4.2.1. Übersicht zu Steganografie-Software	43
4.2.2. Übersicht zu Wasserzeichen-Software	46

4.2.3. Zusammenfassung	47
4.3. Verwendeter Forschungsdatensatzes	48
4.4. Vorstellung von Merkmalen der Stego-Bilder aus dem Forschungsdatensatz	49
4.4.1. Metadatenvergleich	49
4.4.2. Dateigrößenunterschiede	50
4.4.3. LSB-Extraktion	53
4.5. Konfiguration des Detektors	55
4.6. Auswertung des Detektors	57
4.6.1. Auswertung mit binärer Betrachtung	57
4.6.2. Auswertung unter Bezugnahme von Gewichtung	59
4.6.3. Fazit zu den zwei Auswertungen	60
5. Schlussfolgerung	61
6. Ausblick	63
6.1. Allgemein	63
6.2. Forschungsdatensatz	63
6.3. Detektor	64
Bibliografie	65
Tabellenverzeichnis	70
Abbildungsverzeichnis	71
Code-Ausschnittsverzeichnis	72
Anhang A: Sonstiges	73

1. Einleitung

Seit 2500 Jahren werden Informationen in Schrift, Bild oder jeglichem geeigneten Objekt versteckt. Diese Technik wird als Steganografie bezeichnet. Der Begriff leitet sich vom griechischen *steganós* (*στεγανός*) und *-graphia* (*γραφία*) ab, was grob übersetzt für *verborgen* und *Schrift* steht. [1] Inzwischen wird im wissenschaftlichen Umfeld oft die Abkürzung *Stego* verwendet.

Im fünften Jahrhundert nutzten die Griechen im Krieg gegen Persien diese Technik, um die Einnahme durch die Perser zu verhindern. 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*, in welchem er sich erstmals, soweit bekannt, mit Methoden zum Verstecken von Nachrichten in Schrift systematisch beschäftigte. [2]

Insbesondere mit der Entwicklung von Computern nahm Steganografie als Forschungsbereich final Einzug in die Wissenschaft. Großes Interesse kam besonders bei der Einbettung von Informationen in Medien wie Bild- und Audiodaten auf. 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. Neben den steganografischen Verfahren wird auch auf Methoden aus dem Information Hiding eingegangen, welches neben Steganografie, Wasserzeichen sowie die Einbettung von Daten in Strukturelementen von Trägermedien einschließt.

1.1. Motivation

Für Entwickler von Schadsoftware, auch bekannt als Malware, ist es wichtig, im Wettstreit mit den Gesetzeshütern einen Schritt voraus zu sein. Dazu wird inzwischen die Technik der Steganografie immer öfter aufgegriffen. Sogenannte Payloads mit weiterem Schadcode für Multi-Layer-Malware oder Befehle für Command-and-Control-Malware (kurz C&C) können damit in das von der Schadsoftware befallene System eingeschleust werden. Gestohlene Daten wie Passwörter aus dem System können ebenso in Trägermedien eingebettet und ausgeschleust werden, ohne bei IT-Sicherheitssystem wie Firewalls oder Antivirenprogrammen aufzufallen. Die genannten 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. Ein solches Fingerprinting kann vielfältige Informationen zum Angriffshergang, Beschaffenheit und Ursprung der Malware wie auch zur Herkunft der damit operierenden Täter liefern. Dies kann für die strafrechtliche Verfolgung von großer Bedeutung sein.

Neben der Verwendung durch Malware wird Steganografie für das Einbetten von digitalen Wasserzeichen genutzt. Bilddateien werden hierbei mit einem sichtbaren oder versteckten Code versehen werden, um ungewünschte Distributionen oder Manipulationen des Bildes nachverfolgen zu können. So können Urheberrechte geschützt und deren Bruch durch beispielsweise Piraterie geahndet werden. Die Analyse von potenziell mit Wasserzeichen gekennzeichneten Bildern kann also wie bei Malware Aufschluss auf Herkunft und konkrete Art der Einbettung liefern. Die Robustheit des verwendeten Algorithmus kann auf diese Weise getestet werden. Außerdem muss

gewährleistet werden, dass bei der Einbettung keine privaten Informationen preisgegeben werden. [5]

Aus dieser Motivation heraus wurden im Rahmen der Erstellung des Exposés zu dieser Arbeit unter anderem die folgenden Forschungsaspekte identifiziert:

- Literatur-Recherche von aktuellen und relevanten Information Hiding Verfahren
- Literatur-Recherche zur Identifikation potenziell geeigneter Ansätze zur Eigenschaftsbestimmung
- Erstellung und Dokumentation eines Merkmalssets zur Evaluation
- Experimentelle Untersuchung des Vorhandenseins dieser Merkmale in Testdatensets

Daraus ergeben sich die im nächsten Abschnitt vorgestellten Ziele dieser Arbeit.

1.2. Zielsetzung

Die Zielstellung dieser Arbeit konzentriert sich auf die folgenden Punkte:

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

Durch das Erreichen der genannten Ziele soll ein erster Ansatz zum Fingerprinting von Stego-Tools basierend auf deren verwendeten Bildern erarbeitet werden. Der Prototyp soll für den Einsatz mit Malware als auch mit Wasserzeichen weiter ausbaubar sein, wobei dies nicht mehr im Fokus dieser Arbeit liegt.

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

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

1.3. Forschungsfragen und Hypothesen

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

Frage: Können Werkzeuge zur Einbettung von Daten mittels Steganografie nur anhand ihrer Ausgabebilder und der Originalbilder 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 Eigenschaften der verwendeten Stego-Algorithmen und technischen Umsetzung zwischen Malware, Stego- und Wasserzeichen-Tools?

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

Frage: In welchen Eigenschaften unterscheiden sich die drei Kategorien?

Hypothese: Bei Malware liegt ein größerer Wert auf Performance als auf Robustheit, während Wasserzeichen-Tools umgekehrt gewichtet sind. Stego-Tools sind je nach Gewichtung der Entwickler aufgestellt.

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

1.4. Aufbau der Arbeit

Im [folgenden Kapitel](#) werden die theoretischen Grundlagen kurz dargelegt. Da im Rahmen dieser Arbeit keine eigenen steganografischen Algorithmen entwickelt werden, wird nicht tiefergehend auf die mathematische Basis eingegangen. Im Kapitel [Methodik](#) werden das Vorgehen zur Erreichung von **Z.1** bis **Z.4** dokumentiert. Das Kapitel [Ergebnisse](#) stellt die gesammelten Ergebnisse mithilfe von Grafiken und Code-Ausschnitten vor. Die Kapitel [Zusammenfassung](#) und [Ausblick](#) geben schließlich einen kurzen Rückblick auf die Arbeit und zeigen weitere Ausbaumöglichkeiten des entwickelten Programmcodes sowie Richtungen zur Forschung auf.

2. Theoretische Grundlagen

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

2.1. Information Hiding

Die Methodik des Information Hiding beschreibt allgemein das Verstecken von Daten in anderen Daten. In das *Cover*-Medium werden Daten eingebettet, wodurch ein verändertes Medium mit den 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 gehören die Techniken der *Steganografie*, des *Watermarkings* (im weiteren Text als Wasserzeichen benannt) und die *Kryptografie*. 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

Die Einbettung in die Struktur ist im Allgemeinen spezifisch auf das Format angepasst. In ein Bild im JFIF-Format [8] lässt sich beispielsweise eine Nachricht an Stelle eines Vorschaubildes (oder *Thumbnail*) einbetten. Außerdem zählen zu dieser Art von Information Hiding das Anfügen von Daten nach einem End-Marker des Dateiformats wie dem EOI-Marker in JPEGs. Die oftmals simple Natur dieser Verfahren erlaubt eine schnelle Implementierung bei hoher Kapazität an Einbettungsdaten. So können zum Beispiel beliebig große Nachrichten an ein JPEG angefügt werden. Lediglich beim Warten für die Anzeige des Bilds oder bei der Übertragung kann die Größe auffallen.

Bei der Steganografie liegt der Fokus auf dem effektiven Verstecken der Existenz von eingebetteten Daten, sodass nur die versendende und die empfangende Person von der Nachricht wissen und diese auch nur entschlüsseln können. Wie bereits in der Einleitung beschrieben, wurde diese Technik schon lange vor digitalen Systemen eingesetzt. [7]

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

2.1.2. Kriterien

Die Techniken des Information Hiding werden nach drei Kriterien eingeordnet:

Kriterien für das Information Hiding [9]

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

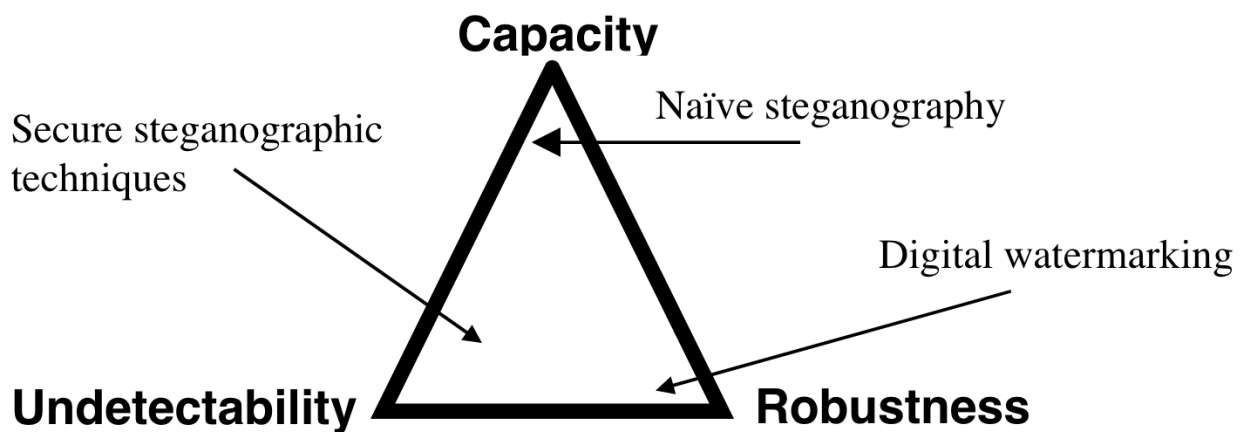


Abbildung 1. Verteilung von Kapazität, Robustheit und Nicht-Detektierbarkeit [10]

Zu den naiven Verfahren können die Struktureinbettung und einfache Stego-Algorithmen zugeordnet werden. Die sichereren Stego-Verfahren weisen eine geringere Kapazität bei höherer Unerkennbarkeit auf. Wasserzeichen-Verfahren haben einen klaren Fokus auf Robustheit, wobei Kapazität und Unerkennbarkeit eine untergeordnete Rolle spielen. Dies ist darin begründet, dass Wasserzeichen üblicherweise nur Identifikationsinformationen beinhalten.

2.1.3. Verfahren

Die klassischen Verfahren der Steganografie 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. Die gezielte Manipulation der darin enthaltenen Werte erlaubt es, dort 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 kaum zu entdecken. Neben des am wenigsten relevanten Bits kann auch der am zweitwenigsten oder drittwenigsten, usw. relevante Bit zur Einbettung genutzt werden. Dies führt zu einer Erhöhung der Kapazität bei einer immer leichteren Detektierbarkeit. 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 dem ursprünglichen entspricht.

XOR-Methode

Ähnlich dem LSB-Verfahren werden XOR-Operationen auf die am wenigsten relevanten Bits der Pixel angewendet. Ein Vorteil gegenüber LSB-Verfahren ist, dabei die Nachricht nicht einfach durch Extraktion direkt lesbar ist. Eine Möglichkeit der Implementierung wird von Joshi und Yadav in [11] vorgestellt.

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 das Cover-Bild in mehrere Blöcke unterteilt wird. In diese Blöcke wird die Einbettung ausgeführt. 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 Modell-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. Durch Forensik-Programme ist dies jedoch einfach zu erkennen.

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 im Zusammenhang mit steganografisch manipulierten Daten. Damit ist sie der sogenannten *Attribution* zugehörig. Diese beschäftigt sich grundsätzlich mit dem Ziel, den oder die Verursachenden von böswilliger Cyberaktivität wie zum Beispiel das Eindringen in Rechnersysteme zu identifizieren. Dabei kann es zunächst um die Rückverfolgung des Angriffs auf die von den Angreifenden verwendeten Systeme gehen. Daraus können dann Rückschlüsse wie die Herkunft der böswilligen Akteure gezogen werden. Oftmals werden Angriffe im Auftrag einer Drittpartei durchgeführt, die möglicherweise über die Angreifer identifizierbar ist. [12]

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 erhärteter Verdacht auf den Einsatz von Stego durch vorherige Schritte der Steganalyse festgestellt wurde. Je nach erwartetem Stego-Verfahren werden unterschiedliche Analyseverfahren angewendet und miteinander kombiniert, um die Einbettungssoftware so detailliert wie möglich zu identifizieren. Konkrete Methoden werden in den [Ergebnissen](#) vorgestellt.

2.2.1. Metriken

Für die Auswertung des Fingerprintings von Stego-Tools werden die folgenden drei Metriken

verwendet, welche unter anderem in [13] beschrieben sind.

Genauigkeit (*Accuracy*)

Beschreibt die Rate der korrekten Zuordnungen im Vergleich zur Gesamtzahl an Bildpaaren.

Formel: $\frac{tp + tn}{tp + fp + tn + fn}$

Präzision (*Precision*)

Beschreibt die Rate der korrekt zugeordneten Bilder zu den insgesamt zu einem Stego-Tool zugeordneten Bildern.

Formel: $\frac{tp}{tp + fp}$

Sensitivität (*Recall*)

Beschreibt die Rate der korrekt zugeordneten Bilder zu den insgesamt korrekten Positiven und Negativen für ein Stego-Tool.

Formel: $\frac{tp}{tp + fn}$

Dabei stehen die Variablen für:

Tabelle 1. Wahrheitsmatrix mit TP, FP, FN und TN

	Echte Positive	Echte Negative
Erwartete Positive	Wahr-Positive / <i>True positive</i> (<i>tp</i>)	Falsch-Positive / <i>False positive</i> (<i>fp</i>)
Erwartete Negative	Falsch-Negative / <i>False negative</i> (<i>fn</i>)	Wahr-Negative / <i>True negative</i> (<i>tn</i>)

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 renommierten 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*: Des Weiteren wurde durch experimentelle Validierung die Funktion des Detektor-Prototyps für **Z.4** nachgewiesen. Dabei wurde der Detektor auf alle Forschungsdaten angewendet und die Ergebnisse stichprobenartig manuell gegengeprüft.

In den nachfolgenden Unterkapiteln werden die verwendeten Suchmaschinen benannt, das manuelle und automatisierte Vorgehen bei der Literaturrecherche zu Malware-Vorkommnissen, das Vorgehen zur Recherche der Stego- und Wasserzeichen-Tools sowie der Forschungsdaten beschrieben und schließlich auf die Entwicklung des Detektor-Prototyps eingegangen.

Alle folgenden Code-Ausschnitt können unter <https://github.com/fabianloewe/master-thesis/releases/tag/v1> heruntergeladen und ausprobiert werden.

3.1. Suchmaschinen

Die manuellen Recherchen wurden über die folgenden Suchmaschinen durchgeführt, insofern 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 renommierten 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 Scholar* und *Google*: 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:

1. Aufbau einer Übersicht zu Malware-Vorkommnissen aus wissenschaftlichen Publikationen und technischen Berichten
2. Automatisierte Sammlung von Malware-Vorkommnissen aus öffentlichen Datenbanken (ohne Analyse von Malware-Samples)

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 mit allgemeinen Analysen zu suchen. Diese boten zu Beginn eine fundierte Übersicht, während Analysen einzelner Malware im späteren Verlauf relevant wurden. 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 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 sowie Forschung hin zu Stego-Malware erkennen. In [\[14\]](#) werden 10 Schadprogramme vorgestellt, die Steganografie oder Information Hiding verwenden. 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 Verlauf der Recherche weiterverwertet wurde. [\[15\]](#)

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

Im technischen Bericht [\[17\]](#) werden sechs Schadprogramme mit ihrer konkreten Funktionsweise an einem Beispiel beschrieben. Darin wird wie auch in [\[18\]](#) die *MITRE Att&ck*-Wissensbasis referenziert. In jener wird eine systematische Zuordnung von sogenannten Techniken und Taktiken, die von Malware eingesetzt werden, zu Malware-Vorkommnissen durchgeführt. Unter den Techniken *T1001.002: Data Obfuscation: Steganography* und *T1027.003: Obfuscated Files or Information: Steganography* ist der Einsatz von Steganografie bei Malware-Vorkommnissen dokumentiert. [\[19\]](#)

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 bereitstellt. [\[20\]](#)

3.2.2. Automatisierte Sammlung aus Datenbanken

Um eine umfängliche Übersicht zu schaffen, wurden daraufhin die Datenbanken von MITRE, 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 Ergebnisliste zu erzielen.

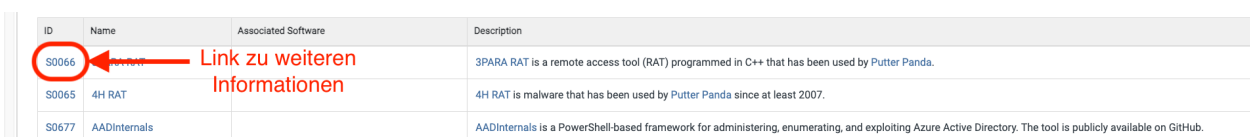
Die Automatisierung wurde mithilfe eines Jupyter Notebooks durchgeführt. Dabei handelt es sich um eine interaktive, lokal ausführbare Plattform, mit welcher Datenanalysen und -visualisierung mit direktem Feedback mittels Python oder anderen Programmiersprachen umgesetzt werden können. Die folgenden Python-Bibliotheken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

- *Selenium*: Ermöglicht das automatische Öffnen von und Interagieren mit Webseiten über einen lokalen Webbrowser [21]
- *Pandas*: Ermöglicht eine verbesserte Datenverarbeitung von tabellarischen Daten [22]
- *requests*: Bietet eine einfache Schnittstelle zum Herunterladen von Daten [23]
- *bibtexparser*: Ermöglicht das Parsen von BibTex-Dateien [24]

Außerdem wurde die Erweiterung *jupyter-ai* [25] für Jupyter genutzt, um mit generativer Text-KI der Firma OpenAI einen ersten Entwurf eines Web-Scrapers zu generieren. Die Aufgabe eines Web-Scrapers ist das Sammeln (oder Schürfen) von Informationen aus Webdaten, vornehmlich HTML-Dateien. Der Web-Scraper wurde dann vom Author individualisiert und zu großen Teilen angepasst.

MITRE-Datenbank

In [Abbildung 2](#) sind Teile der Software-Tabelle der MITRE-Datenbank zu sehen, deren Links nun gesammelt und nach Steganografie-Malware gefiltert werden sollen.



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 [26]

Im Folgenden wird ein Code-Ausschnitt dargestellt, in dem ein Web-Scraper automatisiert die benannte Tabelle nach den relevanten Daten absucht.

Code-Ausschnitt 1: Web-Scraper für Software-Tabelle der MITRE-Datenbank

```
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(...)`. Darin wird zunächst mittels des Webrivers der Link zur Software-Seite von MITRE Att&ck geöffnet. Über einen CSS-Selektor wird in den beiden Zeilen 28 und 29 die Tabelle mit den allen gelisteten Softwares ausgewählt und die gesammelten Links aus den `href`-HTML-Attributen extrahiert.

Es wird über die Links iteriert und gewartet, bis der Hauptteil der Seite im Browser geladen wurde. Dann wird 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 Absturz Speicher konsumiert.

Abbildung 3 zeigt ausschnittsweise, wie Software auf der MITRE-Webseite angezeigt wird. Markiert sind die Datenpunkte, die extrahiert werden sollen.

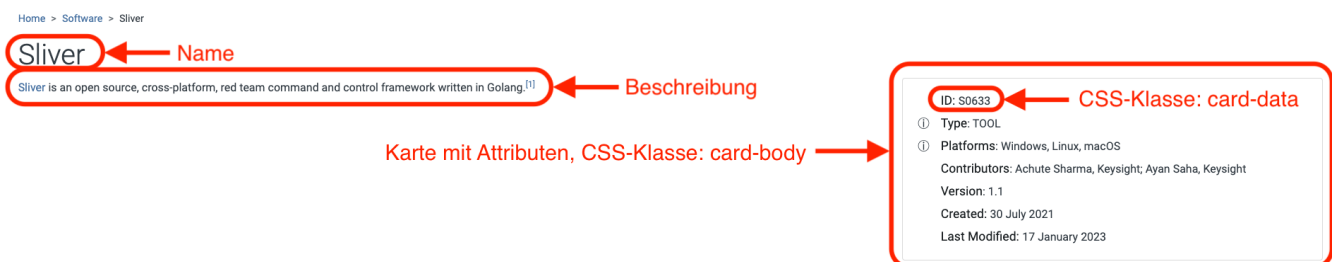


Abbildung 3. Beispiel einer Software auf der MITRE-Webseite [27]

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

Code-Ausschnitt 2: Web-Scraper für einzelne Software-Seiten der MITRE-Datenbank

```
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` nach dem Namen 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 der 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 3](#) 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 4](#) 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
		.004 Application Layer Protocol: DNS	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 Encrypted Channel: Asymmetric Cryptography	Sliver can u
Enterprise	T1041	Exfiltration Over C2 Channel	Sliver can e

CSS: sub

Abbildung 4. Beispiel einer *Techniques Used*-Tabelle [27]

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 dem Tabellenkopf

2. Die Zeile hat die CSS-Klasse `noparent` und es sind 4 Spalten mit einer geteilt in zwei ⇒ 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 ⇒ Übernehme die Daten der ersten beiden Zellen aus der vorherigen Zeile, übernehme die restlichen Zellen

Abschließend wird die `ID` in `MITRE ID` umbenannt, um das Dictionary mit den Techniken zusammenzuführen.

Anzumerken ist dabei, dass die beiden Web-Scraper in einer optimierten Version zusammengefasst werden sollten, sodass die Software-Seiten nicht zweimal 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 eine 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 Stego-Malware im Bildbereich in der Malware-Datenbank *Malpedia* des Fraunhofer FKIE zu suchen.

Code-Ausschnitt 3: Web-Scraper für Malpedia-Datenbank

```
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 @property="og:title"]').get_attribute(
18                'content')
19        except NoSuchElementException:
20            pass
21
22        # The description is not always available
23        try:
24            # Look for a meta tag that contains "description" in the name or property attribute ④
25            description = driver.find_element(By.XPATH,
26                '//meta[@name="description" or contains(@property, "description")]')
27        ).get_attribute('content')
28        except NoSuchElementException:
29            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
36                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 Suche nach dem Wort **steganography** im Text der Webseite in Kleinbuchstaben
- ③ Suche nach einem **meta**-HTML-Element mit dem Attribut **name** gesetzt auf **title** **oder** dem Attribut **property** gesetzt auf **og:title**
- ④ Suche nach einem **meta**-HTML-Element mit dem Attribut **name** gesetzt auf **description** **oder** dem Attribut **property**, dessen Wert **description** enthält
- ⑤ Suche 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 Suche nach den Namen von Plattformen, die in der Konstante **PLATFORMS** definiert sind, welche 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 und 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, die angegriffenen Plattformen der im Artikel beschriebenen Malware 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. Ein anderer Ansatz wäre jeden einzelnen der mehr als 10000 Einträge mit dem Web-Scraper durchzugehen, wobei mit einer sehr hohen Laufzeit zu rechnen ist.

```
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 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, wie im folgenden Code-Block zu sehen ist.

Code-Ausschnitt 5: Extraktion der Links von stego-in-the-wild

```
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('*'), lines)
6     list_entries = itertools.takewhile(lambda line: line.startswith('*'), list_entries)
7
8     # Extract links and their descriptions ②
9     links = []
10    for entry in list_entries:
11        link, description = entry.split(':', 1)
12        name, url = link.split('](', 1)
13        name = name[3:]
14        url = url[:-1]
15        links.append((name, url, description.strip()))
16    return links
```

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 vom Repository-Autoren 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.

Code-Ausschnitt 6: Zusammenführung und Bereinigung der gesammelten Malware-Vorkommnisse

```
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 Attack 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
50 ④
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 Versuche, 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 Entferne alle Einträge, die *404* und *Not Found* im Namen oder der Beschreibung beinhaltet, da diese nicht valide sind
- ④ Säubere die Datumswerte
- ⑤ Füge die Plattformen als Text mit Kommata getrennt zusammen
- ⑥ Füge die Techniken als Text zusammen, die mit Steganografie zutun haben
- ⑦ Füge die Referenzen als Text mit Kommata getrennt zusammen
- ⑧ Entferne 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 mit Malware-Namen im Artikel-Namen durchgegangen und dem Malware-Eintrag als Referenz beigefügt. Anschließend findet die Datenbereinigung statt.

Das finale Ergebnis wird im [Abschnitt 4.1](#) ausgewertet, woraus die Anforderungen an die folgende Literaturrecherche abgeleitet wurden.

3.3. Literaturrecherche zu Steganografie- und Wasserzeichen-Tools

Basierend auf der zuvor erfolgten Sammlung an Malware wurde die Recherche mit den nachfolgenden Anforderungen 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 der Quellcode muss verfügbar sein, damit die Software einer Kategorie entsprechend den [theoretischen Grundlagen](#) zugeordnet werden kann.

3. **Benutzbarkeit:** Die Software muss nach heutigem Stand in maximal drei Schritten (zum Beispiel drei Kommandozeilenbefehle) nutzbar sein. Zu veraltete oder zu kompliziert anzuwendende Programme werden als nicht mehr relevant betrachtet werden.

Des Weiteren muss die Software eine Form des Information Hiding implementieren, das sich bei den Malware-Vorkommnissen wiederfinden lässt oder als erwartbare Weiterentwicklung jener angesehen werden kann.

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 [28] und [29] sowie von Virma et al. in [30]. 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 [31] 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 Stego-Apps bezeichnet werden, wie die Arbeiten von Chen et al. [32] und [33] sowie von Newman et al. [34] 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 [Abschnitt 4.2.1](#) zu finden.

3.3.2. Wasserzeichen-Software

Wie in den [theoretischen Grundlagen zu Verfahren des Information Hiding](#) 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. Fokussiert wurden Publikationen, die einen Überblick über die existierenden Algorithmen verschaffen oder solche Algorithmen attackieren. Für diese wird im Allgemeinen auf bekannte, öffentlich zugängliche Tools zurückgegriffen.

Im [Abschnitt 4.2.2](#) werden die in den Publikationen [35, 36, 37] 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 Eingabe-Formular das weitere Konfigurieren des gewünschten Datensatzes.

Für den Android-Bereich stehen fünf und für den iOS-Bereich sechs verschiedene Smartphone-Modelle zur Verfügung, auf denen Bilder aufgenommen wurden. Für Android gibt es fünf und für iOS lediglich eine Steganografie-App zur Auswahl. 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. Damit ist der statistischen Auswertbarkeit der Daten Genüge getan. Zudem besteht die Verwendung der StegoAppDB darin, dass für die Steganografie-Apps keine Cover-Stego-Bildpaare mehr generiert werden müssen und so der Detektor-Prototyp direkt auf diesen Daten angewendet werden kann. Die in den Stego-Apps verwendeten Algorithmen basieren größtenteils auf LSB-Verfahren und sind damit mit dem aktuellen Stand und Zukunftstrend der Malware vergleichbar, die ebenfalls tendenziell mehr auf LSB-Verfahren setzen.

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

3.5. Implementierung eines Prototyps zum Fingerprinting steganografischer Verfahren in Bilddaten

Zur Umsetzung eines prototypischen Detektors für die Identifikation bestimmter Steganografie-Werkzeuge wurde grundsätzlich in drei 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 und Ableitung von Regeln basierend auf gefundenen Merkmalen zur Abarbeitung für den Detektor

3.5.1. Merkmalskategorien

Aus den vorgestellten [Verfahren zur Einbettung](#) lassen sich Merkmalskategorien ableiten, welche 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.

Tabelle 2. Verwendete Analyse-Tools zur Implementierung des Detektor-Prototyps

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 sowie Angriffe auf Steganografie mittels maschinellen Lernen

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. Zudem wurden die nachfolgend beschriebenen Analyseverfahren als Kommandos ergänzt.



Weitere nützliche Tools 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.

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. Die folgende Tabelle zeigt diese Analyseverfahren.

Tabelle 3. Ausgewählte Analyseverfahren zur Implementierung des Detektor-Prototyps

Name	Domäne	Blind? (Genügt Stego-Bild)	Ziele
Metadatenvergleich	Struktur	Nein	<ul style="list-style-type: none">• 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

Name	Domäne	Blind? (Genügt Stego-Bild)	Ziele
Dateigrößenunterschied	Struktur	Nein	<ul style="list-style-type: none"> • Analyse der Veränderung der Dateigröße zwischen Cover- und Stego-Bild • Herausarbeiten möglicher Zusammenhänge oder identifizierbarer Intervalle in der prozentualen Veränderung
LSB-Extraktion	Raum	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • Identifikation der verwendeten Parameter zur LSB-Einbettung. • Identifikation möglicher eingebetteter Signaturen, Längenangaben der Nachricht, etc.

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

Metadatenvergleich

Pro Stego-App:

1. Lese Metadaten von 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.

Code-Ausschnitt 7: Implementierung des Metadatenvergleichs

```
def exif(filename) -> dict:
    """Return the EXIF data of an image."""
    with ExifTool() as exif:
        return exif.get_metadata(str(filename))
```

```
# }}}

def metadata_diff(cover, stego) -> dict:
    """Return the differences between the metadata of two images."""
    cover_data = exif(cover)
    stego_data = exif(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. Dabei entspricht `cover_data` `c`, `stego_data` `s` und `diff` `u`. 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.

Code-Ausschnitt 8: Implementierung des Dateigrößenunterschieds

```
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. Dabei ist `c` als `cover_size`, `s` als `stego_size` und `d` als `diff` eingesetzt. 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. Zudem kann bestimmt werden, ob die Pixel zeilen- oder spaltenweise durchgegangen werden sollen. Daraus ergeben sich die Parameter Anzahl der LSBs *n* als *bits*, verwendete Farbkanäle *channels*, das Endian-Format *endian* und die Iterationsrichtung *direction*.
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.

Code-Ausschnitt 9: Implementierung der LSB-Extraktion

```
def lsb_extract(input_image, bits=1, channels='RGB', endian='little', direction='row') -> np.ndarray:
    """Extract a message from an image using the LSBs method and returns it.

    :param input_image: the input image
    :param bits: the number of bits to extract
    :param channels: the channels to extract the message from. Options: 'R', 'G', 'B', 'A' or a combination of
    them. **Default**: 'RGB'
    :param endian: the endianness of the message. Options: 'little' or 'big'. **Default**: 'little'
    :param direction: the direction in which to traverse the image. Options: 'row' or 'col'. **Default**: 'row'
    :return: the extracted message
    """

    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 >= 8:
            tmp = msg_byte >> 8
            message.append(msg_byte & 0xFF)
            msg_byte = tmp
            shift -= 8
    return np.array(message, dtype=np.uint8)

_COL_MAP = {'R': 0, 'G': 1, 'B': 2, 'A': 3}

def _load_image(img_path: Path, convert_mode='RGB', channels=None, direction='row'):
    if 'A' in channels:
        convert_mode = 'RGBA'

    with Image.open(img_path) as img:
        arr = np.array(img.convert(convert_mode))

    if direction == 'col' or direction == 'column':
        arr = arr.transpose(1, 0, 2)

    channels = [*channels] if channels else None
    if (convert_mode == 'RGB' and 0 < len(channels) < 3) or (convert_mode == 'RGBA' and 0 < len(channels) <
4):
        arr = arr[:, :, [_COL_MAP[c] for c in channels]]
    return arr.reshape(-1)

def _extract_message(img_path: Path, convert_mode='RGB'):
    data = _load_image(img_path, convert_mode, channels, direction)
    if bits == 1 or bits.bit_count() == 1:
        if endian == 'big':
            return _extract_bits_opt_big(data)
        else:
            return _extract_bits_opt_little(data)
    else:
        if endian == 'big':
            return _extract_bits_big(data)
        else:
            return _extract_bits_little(data)

return _extract_message(input_image)

```

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 allgemeine Versionen zur Extraktion verwendet. `np.zeros()` sowie `np.array()` stammt aus der `numpy`-Bibliothek [38].

Die Umsetzung der Schritte 2 bis 4 erfolgte in dieser Arbeit mittels Regeln für den [Detektor](#).

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 können in Regeln für den Detektor, wie im nächsten Abschnitt beschrieben, aufgerufen werden.

3.5.4. Implementierung des Detektor-Prototyps

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-Scannern fiel die quelloffene Lösung YARA des Malware-Analyse-Portals VirusTotal als möglicher Detektor für diese Arbeit auf. [39]

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

Code-Ausschnitt 10: 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 ein 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 zur Umsetzung der zuvor vorgestellten Analysen einsetzbar 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 darauf zugegriffen werden kann. Diese Funktion ist über selbst entwickelte Module in der Programmiersprache C zwar möglich, wurde aber aufgrund der Komplexität der Programmiersprache selbst sowie des Build-Systems an dieser Stelle nicht weiterverfolgt.

Dennoch diene 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:

Code-Ausschnitt 11: Beispiel einer 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

Die Konfigurationsdatei muss den Schlüssel **isd** mit einem der Werte **"1"**, **"1.0"**, **"1.0.0"** enthalten, um als legitime Konfiguration für den Detektor akzeptiert zu werden. Danach sollten alle zu erkennenden Tools als Liste unter dem Schlüssel **tools** definiert werden. Die Liste besteht aus Objekten mit den folgenden Schlüsseln:

Beschreibung von Stego-Tool-Objekten in der Konfigurationsdatei des Detektor-Prototyps

- **name**: Name des Stego-Tools
- **desc** (optional): Kurze Beschreibung zum Stego-Tool
- **tags** (optional): Tags zur Zuordnung zu frei wählbaren Kategorien. (Die Nutzung dieser Tags zur Sortierung wurde angedacht, aber nicht mehr umgesetzt. Für weitere Ausbaustufen mit komplexeren Darstellungen der Ergebnisse des Detektors sollten die Tags jedoch in Betracht gezogen werden.)
- **version** (optional): Version des Stego-Tools zu Dokumentationszwecken

Der Regelbaum kann einen oder mehrere Einstiegspunkte haben, die unter dem Schlüssel **rules** als Liste definiert werden. Ein Regel-Objekt weist die folgenden Schlüssel auf:

Beschreibung von Regel-Objekten in der Konfigurationsdatei des Detektor-Prototyps

- **name**: Name der Regel
- **desc**: Kurze Beschreibung der Regel
- **tags** (optional): Tags zur Zuordnung zu frei wählbaren Kategorien. (siehe vorheriger Beschreibung von Stego-Tool-Objekten)
- **tools**: List von Objekten mit den Schlüsseln **name** und **weight**:
 - **name**: Name entsprechend einem Stego-Tool aus der obersten **tools**-Liste
 - **weight**: Gibt als eine Zahl größer 0 an, wie gewichtig der Erfolg dieser Regel zum Erkennen des Stego-Tools ist. Die Gewichtung eines Stego-Tools wird beim Durchlaufen des Regelbaums bei jeder erfolgreichen Regel dem vorherigen Wert aufaddiert.
- **match**: Ein Python-Ausdruck als **str** oder ein Objekt mit den Schlüsseln **value** und **cond**. Wird **value** und **cond** verwendet, kann der zu vergleichende Wert im Debug-Modus ausgegeben werden. Sowohl **value** als auch **cond** sind ebenfalls Python-Ausdrücke.
- **next** (optional): Definition der nachfolgenden Regel, die geprüft werden soll, wenn diese erfolgreich war

Schließlich kann eine Regel auch aus mehreren anderen zusammengesetzt sein, was als **CompoundRule** bezeichnet wurde. Diese dienen dem Kontrollfluss beim Durchlaufen des Regelbaums. **CompoundRule**s haben die folgenden Schlüssel:

Beschreibung von zusammengesetzten Regel-Objekten in der Konfigurationsdatei des Detektor-Prototyps

- **name**: Name der Regel
- **desc**: Kurze Beschreibung der Regel
- **tags** (optional): Tags zur Zuordnung zu frei wählbaren Kategorien. (siehe vorheriger Beschreibung von Stego-Tool-Objekten)
- **operator**: Legt fest, wie vorgegangen werden soll, wenn Unterregeln erfolgreich waren. Die folgenden Werte sind erlaubt:
 - **any**: Sobald eine Unterregel in der Reihenfolge ihrer Definition erfolgreich war, gilt diese **CompoundRule** als erfolgreich und der weitere Abarbeitung der Unterregeln wird beendet
 - **all**: Alle Unterregeln müssen erfolgreich sein
 - **none**: Keine Unterregel darf erfolgreich sein
 - **each**: Jede Unterregel wird abgearbeitet. Dabei ist egal, ob die vorherige erfolgreich war oder nicht.
- **rules**: Die Unterregeln dieser zusammengesetzten Regel

Der Detektor stellt zu dem das Python-Modul **attacks** des Tools alethia und das **os**-Modul [40] aus der Standardbibliothek von Python für die Ausdrücke im **match**-Teil der Regeln bereit. Neben den genannten Modulen sind die Variablen **cover** und **stego** verfügbar, welche den Zugang zum Cover- und Stego-Bild erlauben. Beide haben die Felder:

- **path**: Dateipfad zum Bild als Instanz der Klasse **pathlib.Path** [41]
- **image**: Instanz der Klasse **Image** der Bildverarbeitungsbibliothek **Pillow** [42]

Die in dieser Arbeit vorgestellten Analyseverfahren sind wie folgt nutzbar:

- **Metadatenvergleich** → **attacks.metadata_diff(cover, stego)**
- **Dateigrößendifferenz** → **attacks.size_diff(list_of_image_pairs)**
- **LSB-Extraktion** → **attacks.lsb_extract(image, bits, channels, endian, direction)**



Weitere Informationen zu den Parametern sind den verlinkten Erklärungen der Implementierungen zu entnehmen.

Auf diese Weise sind alle weiteren Funktionen wie beispielsweise **spa** des **attacks**-Modul aufrufbar und können in Regeln zur Auswertung herangezogen werden.

Im **Beispiel** wird mit diesen Bausteinen 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.

Der Detektor besteht aus drei Python-Dateien:

- config.py:** Beinhaltet Datenklassen, welche die Konfigurationsdatei abbilden.
- detect.py:** Einstieg in den Detektor, indem die Konfigurationsdatei sowie die Eingabebilder eingelesen und die Regeln aus der Konfiguration evaluiert werden.
- eval.py:** Beinhaltet die **Evaluator**-Klasse, welche die Abarbeitung der Regeln vornimmt.

Zur Ausführung sollte Python 3.10 oder höher genutzt werden. Es sollten zunächst die Abhängigkeiten aus der Datei `requirements.txt` installiert werden. Dafür kann der Befehl `pip install -r requirements.txt` im Verzeichnis `tools/detector` benutzt werden. Außerdem müssen die Abhängigkeiten von aletheia entsprechend der zugehörigen Dokumentation erfüllt werden. Mittels `python detect.py --help` kann dann der Detektor in `tools/detector` ausgeführt werden.

Der Befehl gibt die möglichen Kommandozeilenparameter aus, was dem Folgenden entsprechen sollte.

Code-Ausschnitt 12: Ausgabe der Kommandozeilenparameter des Detektors

```
Usage: detect.py [OPTIONS]

Detects the used stego tool to hide data in the image

The tool uses a config file to define rules for detecting the stego tool.
Cover and stego images can be provided as arguments or read from stdin. If
the images are read from stdin, they should be provided as pairs of paths
separated by a comma. Only the first two values that are separated by a
comma are considered and the rest is currently discarded.

Options:
  -i, --cover-image PATH  Path to a cover image
  -s, --stego-image PATH  Path to a stego image
  --from-stdin             Read cover and stego images from stdin as pairs of
                           paths separated by a comma
  -c, --config FILE        Path to the config file in YAML format [required]
  --aletheia DIRECTORY     Path to the Aletheia root folder
  --help                  Show this message and exit.
```

Mit den Optionen `-i` und `-s` werden die Pfade zum Cover- und zum Stego-Bild angegeben, auf die der Detektor angewendet werden soll. Außerdem ist es mit `--from-stdin` möglich, Komma-separierte Cover-Stego-Bildpaare aus dem Standardinput einlesen zu lassen. Dies kann nützlich sein, um beispielsweise Paare aus einer Datei auszulesen und über die Kommandozeile an den Detektor weiterzuleiten. Mittels `-c` wird die Konfigurationsdatei übergeben. Die Option `--aletheia` muss angegeben werden, wenn sich das gleichnamige Tool nicht im Verzeichnis unter `../tools/aletheia` befindet. Andernfalls schlägt der Import des zuvor benannten `attacks`-Modul fehl.

Die Ausführung des Programms beginnt, indem geprüft wird, ob die Pfade aus `-i` und `-s` beziehungsweise der Paare aus `--from-stdin` auf Bilddateien zeigen. Daraufhin wird die

Konfigurationsdatei eingelesen und die Daten in Python-Klassen zur weiteren Verarbeitung geladen. Schließlich beginnt der **Evaluator** mit der Überprüfung des oder der Cover-Stego-Paar/e anhand der Regeln aus der Konfigurationsdatei. Nachdem alle Paare geprüft wurden, werden die Ergebnisse ausgegeben.

3.5.5. Auswertung des Detektors

Die Auswertung des Detektors wurde mit dem in [Abschnitt 4.5](#) vorgestellten Regeln durchgeführt. Betrachtet wurden zwei Möglichkeiten, Detektionen zu werten:

- Binäre Betrachtung: Jede Detektion mit einer Gewichtung größer als 0 wird als positiv oder andernfalls als negativ eingestuft.
- Einbeziehung von Gewichtung: Nur die höchste Gewichtung unter allen Detektionen für ein bestimmtes Cover-Stego-Paar wird als positiv eingestuft. Bei gleicher Gewichtung wird die Detektion aufgrund mangelnder Eindeutigkeit als negativ angesehen.

Der folgende Code-Ausschnitt zeigt die Berechnung der [Metriken](#) für die binäre Betrachtung in Python.

Code-Ausschnitt 13: Evaluierung des Detektors unter binärer Betrachtung

```
all_detections = [(m, d) for m, detections in data.items() for d in detections]
stats = {}
for method, detections in data.items():
    stats[method] = {
        'true_positives': sum(m == method and d.weights.get(method, 0) > 0 for m, d in all_detections),
        'false_positives': sum(m != method and d.weights.get(method, 0) > 0 for m, d in all_detections),
        'false_negatives': sum(m == method and d.weights.get(method, 0) == 0 for m, d in all_detections),
        'true_negatives': sum(m != method and d.weights.get(method, 0) == 0 for m, d in all_detections)
    }
```

Die folgenden Variablen sind definiert:

Definierte Variablen zur Auswertung des Detektors

- data** ist ein Dictionary, indem für jede Stego-App eine Liste von Detektionen gespeichert ist. Diese Liste wurde für alle Cover-Stego-Bildpaare der tatsächlich zugehörigen Stego-App erstellt. Eine Detektion ist ein Objekt, das der Detektor erstellt hat, wenn eine Regel im Regelbaum erfolgreich war. **Detection**-Objekte beinhalten unter anderem die Gewichtungen der detektierten Stego-Apps im Feld **weights**, was ein Dictionary aus App-Namen und Gewichtung als Zahl ist.
- all_detections** ist eine Liste der gesamten Detektionen. Darin besteht jedes Element paarweise aus der tatsächlichen Stego-App und einer Detektion.
- stats** ist ein Dictionary, indem für jede Stego-App die Metriken gespeichert werden.

Nach Definition der Variablen wird über alle Stego-Apps und ihrer Detektionen aus **data** iteriert. Die Variable **method** beinhaltet den Namen der aktuellen Stego-App und **detections** die

dazugehörigen Detektionen. Zur Berechnung der Werte wird jeweils `all_detections` durchlaufen, wobei `m` die tatsächliche Stego-App und `d` die aktuell zu untersuchende Detektion beinhaltet.

Die Werte werden wie folgt berechnet:

Berechnung der TP, FP, FN und TN bei binärer Betrachtung zur Auswertung des Detektors

- **true_positives** (TP): Summiere alle Gewichtungen auf, bei denen die zu untersuchende Stego-App **method** *gleich* der tatsächlichen Stego-App **m** ist und die Gewichtung der Stego-App in **method** *größer als* 0 ist.
- **false_positives** (FP): Summiere alle Gewichtungen auf, bei denen die zu untersuchende Stego-App **method** *ungleich* der tatsächlichen Stego-App **m** ist und die Gewichtung der Stego-App in **method** *größer als* 0 ist.
- **false_negatives** (FN): Summiere alle Gewichtungen auf, bei denen die zu untersuchende Stego-App **method** *gleich* der tatsächlichen Stego-App **m** ist und die Gewichtung der Stego-App in **method** *gleich* 0 ist.
- **true_negatives** (TN): Summiere alle Gewichtungen auf, bei denen die zu untersuchende Stego-App **method** *ungleich* der tatsächlichen Stego-App **m** ist und die Gewichtung der Stego-App in **method** *gleich* 0 ist.

Analog dazu zeigt der nächste Code-Ausschnitt die Auswertung unter Einbeziehung der Gewichtung.

Code-Ausschnitt 14: Evaluierung des Detektors mit Einbeziehung der Gewichtung

```
def get_max_weight(detection):
    max_weight = max(detection.weights.values() or [0])
    if list(detection.weights.values()).count(max_weight) > 1:
        return None
    return max_weight if max_weight > 0 else None

all_detections = [(m, d) for m, detections in data.items() for d in detections]
stats = {}
for method, detections in data.items():
    stats[method] = {
        'true_positives': sum(
            m == method and d.weights.get(method, 0) == get_max_weight(d) for m, d in all_detections),
        'false_positives': sum(
            m != method and d.weights.get(method, 0) == get_max_weight(d) for m, d in all_detections),
        'false_negatives': sum(
            m == method and d.weights.get(method, 0) != get_max_weight(d) for m, d in all_detections),
        'true_negatives': sum(
            m != method and d.weights.get(method, 0) != get_max_weight(d) for m, d in all_detections)
    }
```

Die **vordefinierten Variablen** sind hierbei unverändert. Die Berechnung von TP, FP, FN und TN geht grundsätzlich genauso wie bei der **binären Betrachtung** vonstatten. Neu ist lediglich die Funktion **get_max_weight(...)**. In dieser wird die höchste Gewichtung von allen in einer Detektion bestimmt. Konnte keine Stego-App erkannt werden, wird **max_weight** auf 0 gesetzt. Kommt die höchste Gewichtung mehrfach vor oder ist die höchste Gewichtung 0, gibt die Funktion **None** zurück. Andernfalls wird **max_weight** zurückgegeben. Mithilfe von **get_max_weight(...)** wird in den Berechnungen geprüft, ob die zu untersuchende Stego-App **method** die höchste Gewichtung in der Detektion **d** erhalten hat.

Die Ergebnisse dieser Auswertungen werden in [Abschnitt 4.6](#) vorgestellt.

4. Ergebnisse

Die folgenden Ergebnisse sind jeweils zu einem Abschnitt in [Methodik](#) zugeordnet. Zunächst werden die gefundenen Malware-Vorkommnisse vorgestellt. Daraufhin werden Programme aus dem Bereich des Information Hiding dargelegt. Es folgt die Vorstellung des Forschungsdatensatzes. Den Abschluss bildet die Auswertung der Ergebnisse des Detektors mit den für den Forschungsdatensatz definierten Regeln.

4.1. Malware-Vorkommnisse im Zusammenhang mit Information Hiding

Der folgende Auszug aus den gesamten gefundenen Malware-Vorkommnissen, bei denen Information Hiding mit Bildern angewendet wurde, zeigt die zehn aktuellsten Funde. Diese wurden erstmals im Zeitraum von Juli 2020 bis Mai 2024 registriert. Die Malware *Zox* und das Framework *Silver* wurden im April 2024 auf MITRE aktualisiert, um ihren aktuellen Stand abzubilden.

Tabelle 4. Auszug aus Malware-Daten

Name	Plattformen	Erstellt am	Zuletzt geändert	MITRE-ID	Medientypen	Stego-Verfahren	Referenzen
requests-darwin-lite	macOS	13.05.2024			PNG	Unbekannt	[43]
APT37	Windows	14.02.2023			JPEG	XOR	[44]
Zox	Windows	09.01.2022	10.04.2024	S0672	PNG	Unbekannt	[45]
Diavol	Windows	12.11.2021	04.12.2023	S0659	BMP	Unbekannt	[46]
ProLock	Windows	30.09.2021	15.10.2021	S0654	JPEG, BMP	Unbekannt	[47]
ObliqueRAT	Windows	08.09.2021	15.10.2021	S0644	BMP	LSB	[48]
Sliver	Windows, Linux, macOS	30.07.2021	11.04.2024	S0633	PNG	Unbekannt	[49]
LiteDuke	Windows	24.09.2020	04.10.2021	S0513	PNG	LSB	[50]
PolyglotDuke	Windows	23.09.2020	26.03.2023	S0518	GIF	LSB	[50]
RegDuke	Windows	23.09.2020	24.03.2023	S0511	PNG	Unbekannt	[50]

Jede der gezeigten Malware wurde für die Ausführung auf dem Betriebssystem Windows entwickelt, wobei *Silver* auch auf macOS und Linux lauffähig ist. Das quelloffene Framework wurde bei einer jüngsten Attacke aus dem Jahr 2024 in das Python-Paket *requests-darwin-lite* eingebunden und so zur Komprimierung von macOS-System eingesetzt. [51]

Fünf Exemplare (*requests-darwin-lite*, *Zox*, *Silver*, *LiteDuke* und *RegDuke*) verwenden PNG-Bilder für ihre Zwecke. Weitere drei (*Diavol*, *ProLock* und *ObliqueRAT*) nutzen Bitmap-Bilder, wobei *ProLock*

auch auf JPEG-Bilder zurückgreift. Das JPEG-Format kommt neben *ProLock* nur beim Angriff der APT37-Gruppe zum Einsatz.

Es zeigt sich, dass von den bekannten Stego-Verfahren dreimal (*ObliqueRAT*, *LiteDuke* und *PolyglotDuke*) eine LSB-Variante implementiert wurde. Der Angriff von APT37 setzte eine XOR-Implementierung ein. Bei den anderen Vorkommnissen konnte kein Verfahren mit Sicherheit bestimmt werden.

Der Ausschnitt aus der vollständigen Tabelle im Verzeichnis `assets/tables/malware-full-data-cleaned-extended.csv` wurde mit den folgenden Schritten erstellt:



1. Gesamttabelle nach Werten in der Spalte `Created` absteigend sortieren
2. Zeilen, deren `Name` dem regulären Ausdruck `["{}@]` entspricht, entfernen. Dabei handelt es sich um nicht-lesbare Einträge, die somit auch nicht weiter verwertet werden können.
3. Ersten 10 Ergebnisse auswählen und die in [Tabelle 4](#) zusehenden Spalten auswählen
4. Englische Bezeichnungen in Deutsch übersetzen

Nachfolgend werden Auswertungen über den gesamten Datensatz der Malware-Vorkommnisse vorgestellt. Als Erstes werden die angegriffenen Betriebssysteme betrachtet.

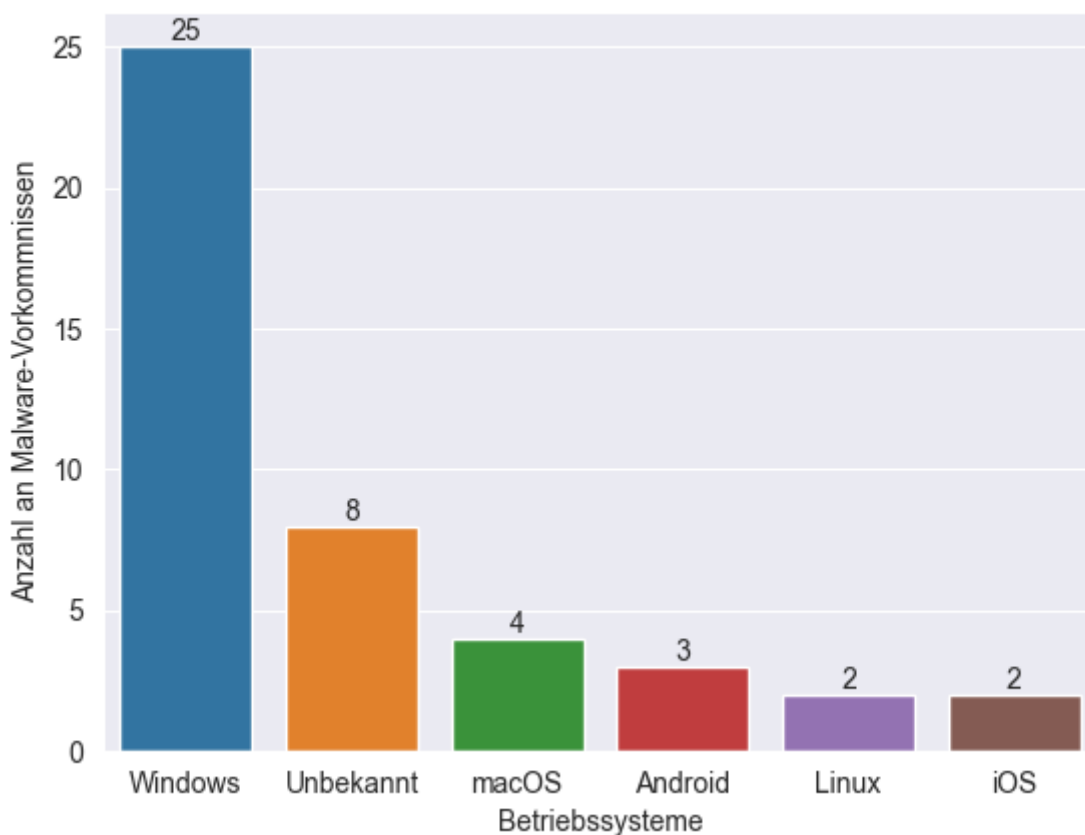


Abbildung 5. Anzahl der Malware-Vorkommnisse pro Betriebssysteme

Windows wurde bei 25 Vorkommnissen, und damit am meisten, als Ziel gewählt. Es folgt macOS bei vier, Android bei drei und Linux und iOS bei zwei böswilligen Programmen. Acht der 36 Vorkommnissen konnten nicht zugeordnet werden.

Der nächste Graph zeigt die Anzahl der Malware-Vorkommnisse pro Medientyp.

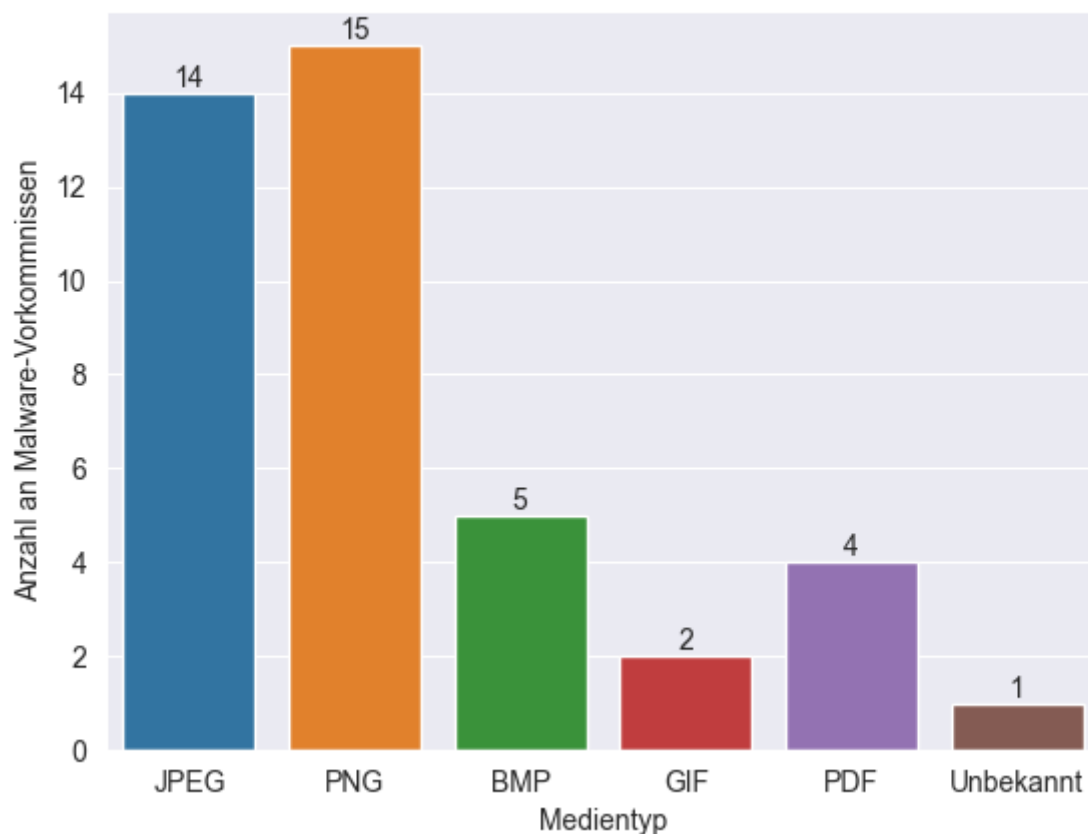


Abbildung 6. Anzahl der Malware-Vorkommnisse pro Medientyp

Dabei zeigt sich, dass sowohl JPEG als auch PNG mit 14 und 15 Malen besonders häufig genutzt wird. BMP-, GIF- und PDF-Dateien sind weniger relevant.

Abschließend ist die Verteilung der Malware-Vorkommnisse bei den Stego-Verfahren zu sehen.

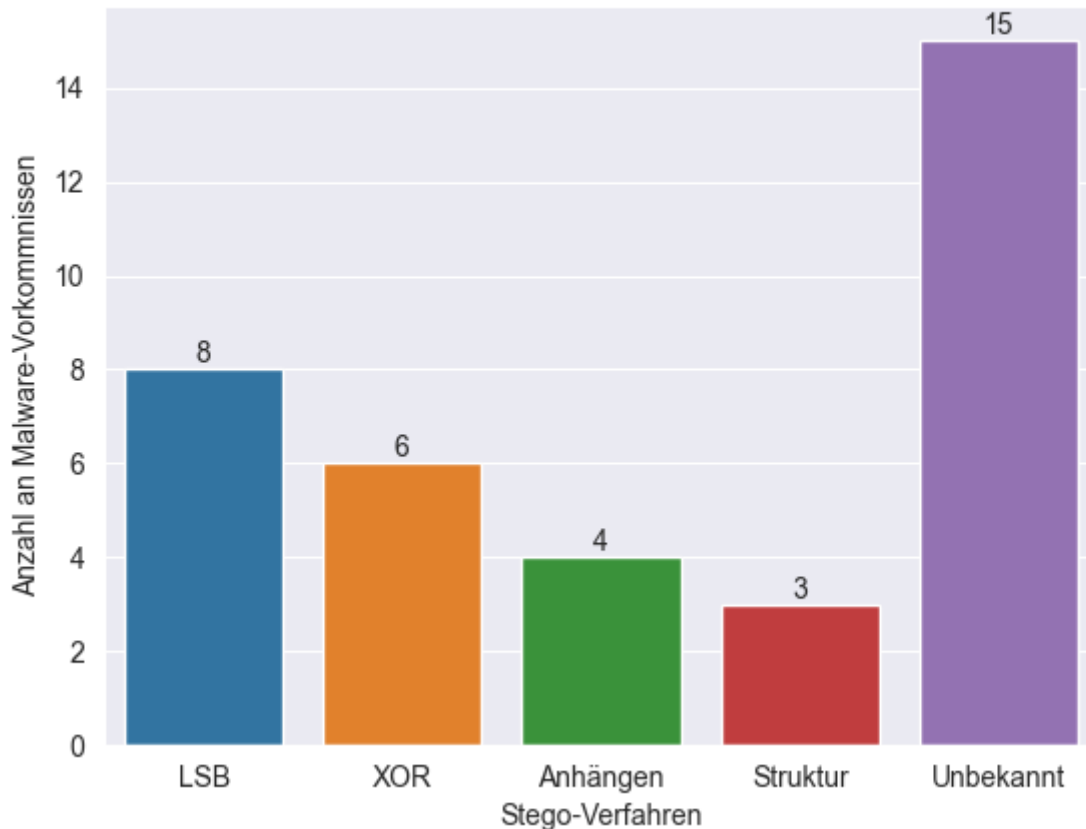


Abbildung 7. Anzahl der Malware-Vorkommnisse pro Stego-Verfahren

Die meisten Vorkommnisse konnten keinem konkreten Stego-Verfahren zugeteilt werden, da oftmals die Artikel keine weitere Auskunft dazu liefern. Dennoch konnten von den bestimmaren Angriffen acht einem LSB-Verfahren zugeordnet werden. Sechs setzten auf XOR-Verfahren, vier hängten ihren Payload ans Ende des Cover-Bilds an und drei fügten ihren Payload in die Metadaten oder an speziell markierten Stellen des Cover-Bilds ein.

4.1.1. Zusammenfassung der Malware-Vorkommnisse

Auch wenn einige Artikel keine genaueren Details zur verwendeten Steganografie liefern, lassen sich klare Tendenzen ablesen. Die meisten Vorkommnisse zielen auf das Betriebssystem Windows ab, verwenden PNG- oder JPEG-Bilder und implementieren eine LSB- oder XOR-Variante. Der Einsatz von Steganografie bei Malware konnte in den letzten vier Jahren mehr als zehnmal verzeichnet werden und ist damit ein aktueller Trend bei Malware-Entwicklern, um nicht durch Antivirensoftware erkannt zu werden.

4.2. Steganografie- und Wasserzeichen-Software

Die anschließenden Unterabschnitte listen die in der Recherche gefundenen Programme auf und vergleichen diese nach ihren Funktionen und ihren erfüllten Anforderungen. Abschließend wird eine Auswahl an Kandidaten vorgestellt.

4.2.1. Übersicht zu Steganografie-Software

In den folgenden Tabellen werden die Steganografie-Programme vorgestellt. [Tabelle 5](#) zeigt die Tools, die auf Desktop-Plattformen wie Windows, macOS und Linux laufen, während [Tabelle 6](#) 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 Stego-Domäne, ihre unterstützten Stego-Algorithmen, die Erfüllung der Anforderungen und schließlich zugehörige Referenzen benannt.

Tabelle 5. Übersicht von Stego-Tools als Desktop-Anwendungen

Name	Plattformen	Domäne	Algorithmen	Anforderung Bekanntheit	Anforderung Funktionsweise	Anforderung Benutzbarkeit	Referenzen
jsteg	Windows, macOS, Linux	Raum	LSB	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als ausführbare Datei verfügbar	[31, 30]
steghide	Windows, macOS, Linux	Raum	LSB	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als ausführbare Datei verfügbar	[31, 30]
stegpy	Windows, macOS, Linux	Raum	LSB	Nicht erfüllt, da zu geringe Bekanntheit	Erfüllt, da quelloffen	Erfüllt, da als Python-Paket nutzbar	[31, 52]
stegify	Windows, macOS, Linux	Raum	LSB	Erfüllt, da über 1000 Bewertungen bei GitHub	Erfüllt, da quelloffen	Erfüllt, da als ausführbare Datei verfügbar	[53]

Name	Plattformen	Domäne	Algorithmen	Anforderung Bekanntheit	Anforderung Funktionsweise	Anforderung Benutzbarkeit	Referenzen
LSB-Steganography	Windows, macOS, Linux	Raum	LSB	Nicht erfüllt, da über 800 Bewertungen bei GitHub	Erfüllt, da quelloffen	Erfüllt, da als Python-Paket nutzbar	[54]
stegolsb	Windows, macOS, Linux	Raum	LSB	Nicht erfüllt, da über 500 Bewertungen bei GitHub	Erfüllt, da quelloffen	Erfüllt, da als Python-Paket nutzbar	[55]
outguess	Windows, macOS, Linux	Frequenz	Outguess	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als ausführbare Datei verfügbar	[31, 30]
f5	Windows, macOS, Linux	Frequenz	F5	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Nicht erfüllt, da keine Dokumentation zur Installation gefunden werden konnte	[31, 30]

Dabei ist zu erkennen, dass sechs der acht Tools auf ein selbst implementiertes LSB-Verfahren setzen und somit in der Raum-Domäne aktiv sind. Lediglich *f5* nutzt den gleichnamigen Algorithmus und *outguess* setzt auf eine eigene Variante in der Frequenz-Domäne. Alle gefundenen Programme sollen auf den drei gängigsten Betriebssystemen Windows, macOS und Linux im Desktop-Umfeld lauffähig sein. Der Entwicklungszeitraum der Tools beläuft sich von 2014 bei *f5* als ältestes bis heute bei *stegolsb*, das weiterhin kleine Aktualisierungen erfährt.

Alle der dargestellten Stego-Tools erfüllen mindestens zwei der drei [Anforderungen](#), wobei *jsteg*, *steghide* und *stegify* diesen vollständig nachkommen. Alle drei Tools setzen LSB-Verfahren um und sind grundsätzlich auf Windows einsetzbar, welches am häufigsten als Angriffsziel gewählt wurde. Damit sind sie geeignete Kandidaten, um Stego-Malware bei der Erstellung des Forschungsdatensatzes zu repräsentieren.

Neben den Desktop-Anwendungen wurden mobile Anwendungen für iOS und Android gefunden.

Diese werden nachfolgend vorgestellt.

Tabelle 6. Übersicht von Stego-Tools als mobile Anwendungen

Name	Plattformen	Domäne	Algorithmen	Anforderung Bekanntheit	Anforderung Funktionsweise	Anforderung Benutzbarkeit	Referenzen
PixelKnot	Android	Frequenz	F5	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da im Google Play-Store verfügbar	[34]
Passlok Privacy	Android	Frequenz		Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da in wissenschaftlicher Arbeit analysiert	Erfüllt, da im Google Play-Store kostenlos verfügbar	[34]
MobiStego	Android	Raum	LSB	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da im Google Play-Store kostenlos verfügbar	[34]
Steganography-Meznik (oder SteganographyM)	Android	Raum	LSB	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da in wissenschaftlicher Arbeit analysiert	Nicht eindeutig erfüllt, da nicht im Google Play-Store verfügbar, aber bei Drittanbietern	[34]
Pictograph	iOS	Raum	LSB	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da in wissenschaftlicher Arbeit analysiert	Erfüllt, da im App-Store kostenlos verfügbar	[34]

Die Stego-Apps sind bis auf bei *PixelKnot* und *Password Privacy* mit einem LSB-Verfahren implementiert. Da Newman et al. in [34] diese Apps analysiert hat, sind die Anforderungen Bekanntheit und Funktionsweise bei jeder gegeben. Bis auf *Steganography-Meznik* können zudem alle Apps einfach auf Endgeräten installiert werden. Als geeignete Kandidaten zeichnen sich daher

insbesondere *MobiStego* und *Pictograph* ab, weil beide die drei Anforderungen erfüllen und LSB-Verfahren wie bei vielen der Malware-Vorkommnisse implementieren.

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 Stego-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.

Die folgende Tabelle stellt Verfahren aus dem aktuellen Stand der Technik dar.

Name	Plattformen	Domäne	Algorithmen	Anforderung Bekanntheit	Anforderung Funktionsweise	Anforderung Benutzbarkeit	Referenzen
DwtDcdSvd	Windows, macOS, Linux	Frequenz	DWT, DCT, SVD	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[56]
StegaStamp	Windows, macOS, Linux	Frequenz	Neuronales Netzwerk	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[57]
SSL Watermarking	Windows, macOS, Linux	Uneindeutigkeit	Neuronales Netzwerk	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[58]
Stable Signature	Windows, macOS, Linux	Uneindeutigkeit	Neuronales Netzwerk	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[59]

Name	Plattformen	Domäne	Algorithmen	Anforderung Bekanntheit	Anforderung Funktionsweise	Anforderung Benutzbarkeit	Referenzen
Tree-Ring Watermarks	Windows, macOS, Linux	Uneindeutig	Neuronales Netzwerk	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[60]
Tree-Ring Watermarks	Android	Uneindeutig	Neuronales Netzwerk	Erfüllt, da in wissenschaftlicher Arbeit verwendet	Erfüllt, da quelloffen	Erfüllt, da als Python-Datei verfügbar	[61]

Es ist zu erkennen, dass alle Verfahren entweder in der Frequenz-Domäne agieren oder aufgrund der Verwendung von maschinellem Lernen auf uneindeutige Weise in die Bilder einbetten.

Zwar erfüllen alle Wasserzeichenverfahren die Anforderungen grundlegend, jedoch nutzt nach aktuellem Stand keine Malware die Frequenz-Domäne oder gar maschinelles Lernen zum Einbetten von Daten. Die erhöhte Robustheit der Verfahren deckt sich damit mit den Annahmen in [Abschnitt 1.3](#) sowie [Abschnitt 2.1.2](#).

4.2.3. Zusammenfassung

Von den Stego-Tools im Desktop-Bereich stellen sich jsteg, steghide und stegify als geeignete Kandidaten heraus, während MobiStego und Pictograph im mobilen Anwendungsbereich hervorstechen. Alle genannten Stego-Programme nutzen LSB-Verfahren, sind auf Windows, Android oder iOS einsetzbar und erfüllen die [Anforderungen](#).

Zusätzlich sollten bis zu zwei Verfahren in der Frequenz-Domäne betrachtet werden, da diese möglicherweise als Weiterentwicklung bei Malware in Betracht gezogen werden könnten. Außerdem sind diese Techniken noch im Bereich der Wasserzeichen im Einsatz, womit diese im Forschungsdatensatz ebenso Beachtung finden würden.

4.3. Verwendeter Forschungsdatensatzes

Wie in [Abschnitt 3.4](#) beschrieben wurde die StegoAppDB als sinnvoller Kandidat für den Forschungsdatensatz gewählt. Die folgenden Vor- und Nachteile sind zu benennen:

Tabelle 7. Vor- und Nachteile der Verwendung der StegoAppDB als Forschungsdatensatzes

Vorteile	Nachteile
Es wurden bereits alle Cover-Stego-Bildpaare generiert und eingebettete Nachrichten sind dokumentiert.	Es sind nur die Stego-Apps enthalten.
Der Datensatz wurde mindestens einmal zuvor bei der Verwendung im Paper [33] getestet.	

Obwohl nur Daten von Stego-Apps enthalten sind, bilden diese jedoch ein gutes Spektrum der [gefundenen Stego-Software](#) ab. Hauptsächlich sind darin LSB-Verfahren wie häufig bei Malware und zweimal Varianten in der Frequenz-Domäne umgesetzt, womit auch Algorithmen ähnlich der Wasserzeichenprogramme untersucht werden können.

Es wurde dabei über eine Auswahlmaske ein Auszug aus der gesamten StegoAppDB benutzt. Die Einstellung des Suchformulars 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 ?

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

☒ Auto Exposure 10 - 7000

☐ Manual Exposure 10 - 7000

Exposure Time

1/11000 - 1/2

1/11000 - 1/2

Number of Images: 24468

Estimated Download Size: 18.53 Gigabyte(s)

Download
Search Again

Abbildung 8. Einstellung des Suchformulars der StegoAppDB für den verwendeten Forschungsdatensatz [34]

Um die vollständigen Cover-Stego-Paare zur Verfügung zu haben, wurden die in der Maske als *pre-stego* bezeichnete Option angewählt. Als Einbettungsprogramme wurden PixelKnot, Passlok

Privacy, MobiStego, PocketStego, Steganography-Meznik und Pictograph ausgewählt. Mit PixelKnot und Passlok Privacy arbeiten zwei der sechs Stego-Apps mit JPEG-Bildern, während der Rest PNG verwendet. 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 15](#))

Der Forschungsdatensatz besteht aus 18570 Bildpaaren. Die gesamte Speichergröße beläuft sich auf 18,53 GB, wobei neben den Cover- auch noch ungenutzte Input-Bilder enthalten sind. Die einzelnen Bilder weisen eine Größe zwischen 100 und 300 KB auf, wurden auf 512x512 Pixel beschnitten und zu Grauwertbildern umgewandelt. Als Einbettungsnachrichten wurden Ausschnitte aus 634 Shakespeare-Stücken zufällig gewählt. [34]

MobiStego, PixelKnot, PocketStego und SteganographyM haben 3060 Bildpaare, Pictograph 4800 und Passlok Privacy 1530. Diese Verteilung konnte in der Auswahlmaske nicht bestimmt werden.

4.4. Vorstellung von Merkmalen der Stego-Bilder aus dem Forschungsdatensatz

In diesem Abschnitt werden Merkmale aus dem Forschungsdatensatz vorgestellt, die als Ergebnisse der [Analyseverfahren](#) abgeleitet werden konnten. Diese werden dann in [Abschnitt 4.5](#) als Regeln definiert.

4.4.1. Metadatenvergleich

Der Metadatenvergleich wurde wie in der Vorstellung der [Analyse von Metadaten](#) durchgeführt. Dabei zeigte sich, dass sich das Metadatenfeld `PNG:SignificantBits` bei von MobiStego veränderten Bildern von `8 8 8` zu `8 8 8 8` änderte. Eine zweite Änderung konnte beim Feld `PNG:ColorType` festgestellt werden, welche sich bei Bildern von MobiStego von `2` auf `6` und bei Bildern von Pictograph von `0` auf `2` beläuft.

Somit lassen sich diese Funde als Merkmale beschreiben.

Abgeleitete Merkmale aus dem Metadatenvergleich

Änderung von `PNG:SignificantBits`

- Wenn von `8 8 8` zu `8 8 8 8` → Hinweis auf MobiStego

Änderung von `PNG:ColorType`

- Wenn von `2` zu `6` → Hinweis auf MobiStego
- Wenn von `0` zu `2` → Hinweis auf Pictograph

4.4.2. Dateigrößenunterschiede

Die folgenden Dateigrößendifferenzen konnten mit der [im Methodikteil vorgestellten Analyse](#) ermittelt werden.

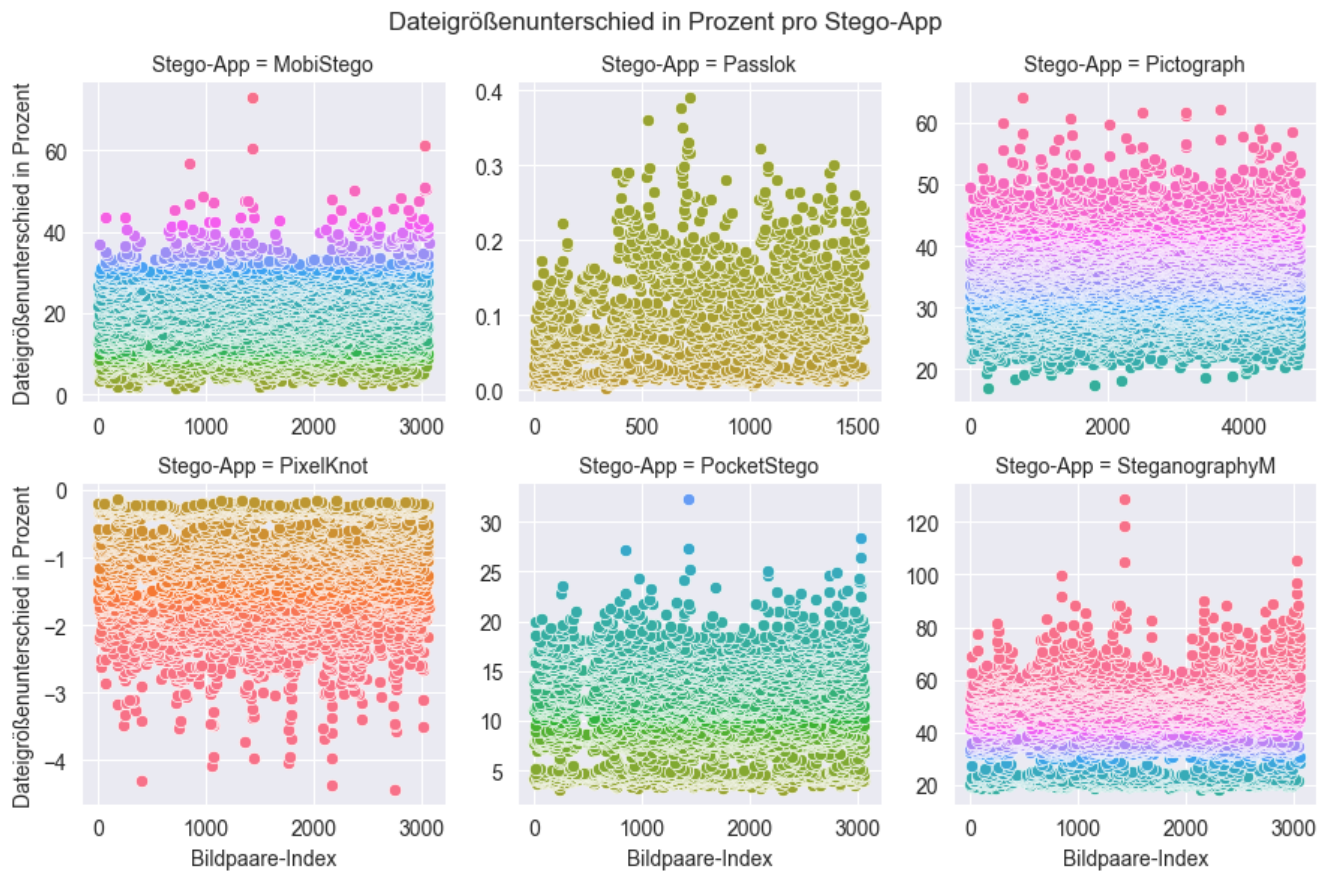


Abbildung 9. Übersicht der Dateigrößenunterschiede als Streudiagramme

Das Diagramm zeigt, dass der Größenunterschied für MobiStego bei 3060 Cover- und Stego-Bildern sich im positiven Bereich zwischen 2 % bis 35 % bewegt. Dabei gibt es einige Ausreißer mit einem Unterschied bis zu 50 %, sehr wenige bis 60 % und lediglich ein Unterschied von 70 % vorkommt. Bei Passlok wurden 1530 Bildpaare untersucht, wobei sich die Verteilung im Bereich von 0,01 % bis 0,25 % mit Ausreißern bis 0,38 % befindet. Für Pictograph standen 4800 Bildpaare bereit, welche einen Unterschied von 16 % bis 50 % mit Ausreißern bis 65 % aufweisen. PixelKnot weist als einzige App bei 3060 Bildpaaren eine Verkleinerung der Dateigröße auf, weshalb sich der Dateigrößenunterschied um -0,01 % bis -3 % mit Ausreißern bis -4,3 % bewegt. PocketStego befindet bei 3060 Bildpaaren im Bereich von 3 % bis 20 % mit Ausreißern bis 33 %. Bei SteganographyM sind die größten Unterschiede in der Dateigröße vernehmbar. Die Unterschiede fallen in den Bereich von 19 % bis 70 % mit Ausreißern bis zu 130 %.

Die Farbgebung ist nach den Werten der Dateigrößendifferenzen auf der Y-Achse gewählt. Gleich oder ähnlich farbige Werte liegen daher über alle sechs Diagramme hinweg nahe beieinander. So zeigt zum Beispiel der Farbverlauf von Pictograph und SteganographyM, dass beide ihren unteren Wertebereich um 20 % in Türkis haben. SteganographyM geht im oberen Wertebereich aus dem Rosafarbenem stark ins Rosarote hinein und verdeutlicht damit dessen höhere Dateigrößendifferenz im oberen Wertebereich im Kontrast zu Pictograph. MobiStego weist eine auffällige blaue und türkise Mitte auf, was erkennen lässt, dass MobiStego im mittleren bis oberen Wertebereich mit Pictograph und SteganographyM überlappt. Der grüne untere Bereich zeigt auf

der anderen Seite, dass es dort keine Überlappung mit Pictograph oder SteganographyM gibt, dafür aber mit PocketStego und womöglich auch mit Passlok. PixelKnot verzeichnet als einzige Stego-App eine Verkleinerung des Dateigrößenunterschieds und weist daher auch einen einzigartigen Farbverlauf aus dem Gelben über Orange ins Feuerrote hinein.

Im nächsten Diagramm ist die Verteilung der prozentualen Dateigrößenunterschiede der Stego-Apps im Vergleich zueinander dargestellt. Die Violinen des Violinendiagramms werden mit der Kerndichteschätzung berechnet. Weitere Informationen dazu sind der Dokumentation in [62] zu entnehmen.

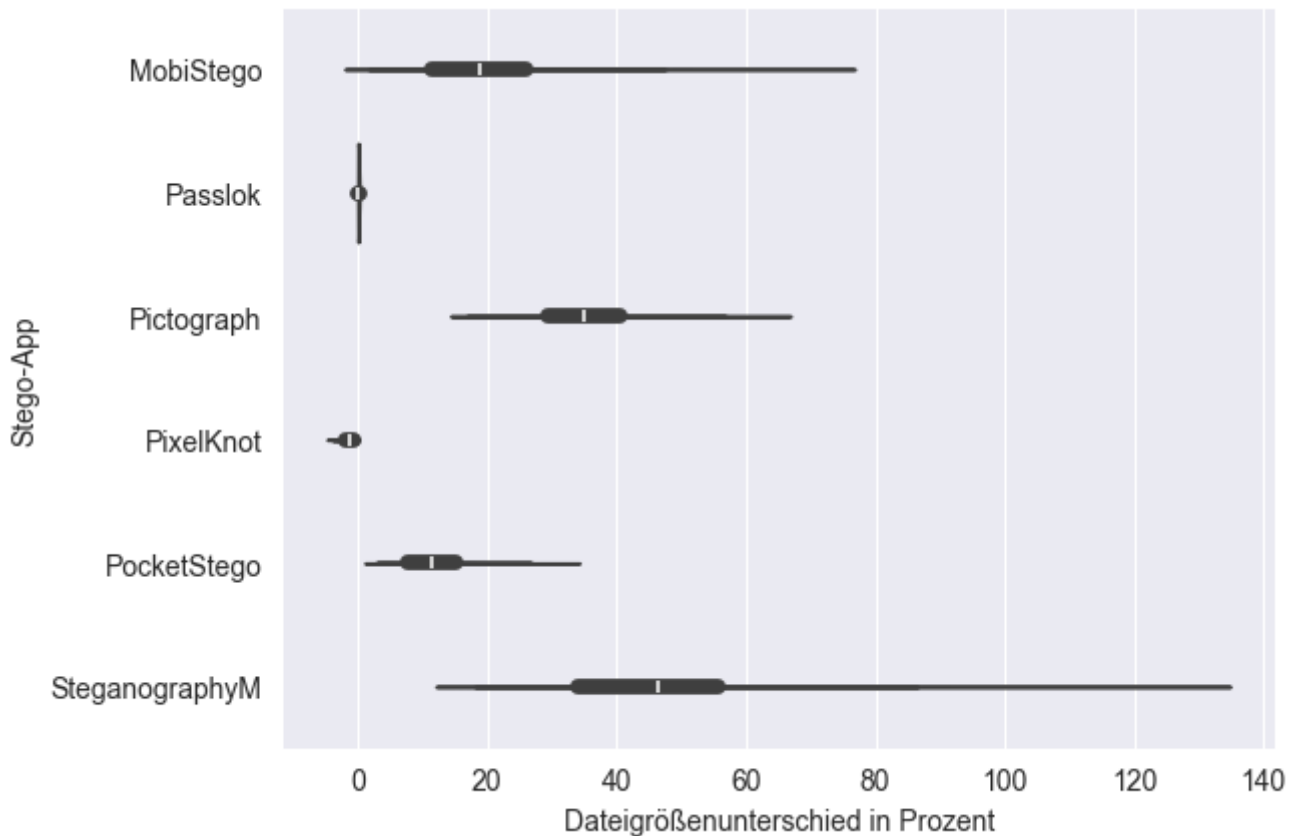


Abbildung 10. Darstellung der Dateigrößenunterschiede mit Ausreißern

Dabei zeigt sich, dass PixelKnot im negativen Bereich keine Schnittmenge und SteganographyM im positiven Bereich ab mindestens 80 % keine Überschneidungen mit anderen Apps aufweist.

Wie in den Diagrammen zu den einzelnen Apps bereits zu sehen war, gibt es am unteren und oberen Ende der meisten Apps einige Ausreißer. Diese unteren und oberen 5 % wurden im folgenden Schnitt ausgeklammert, um Apps mit einer höheren Wahrscheinlichkeit für einen bestimmten Dateigrößenunterschied eher zu detektieren als die Ausreißer weniger wahrscheinlicher Apps. Nach dieser Bereinigung sieht die Verteilung wie folgt aus:

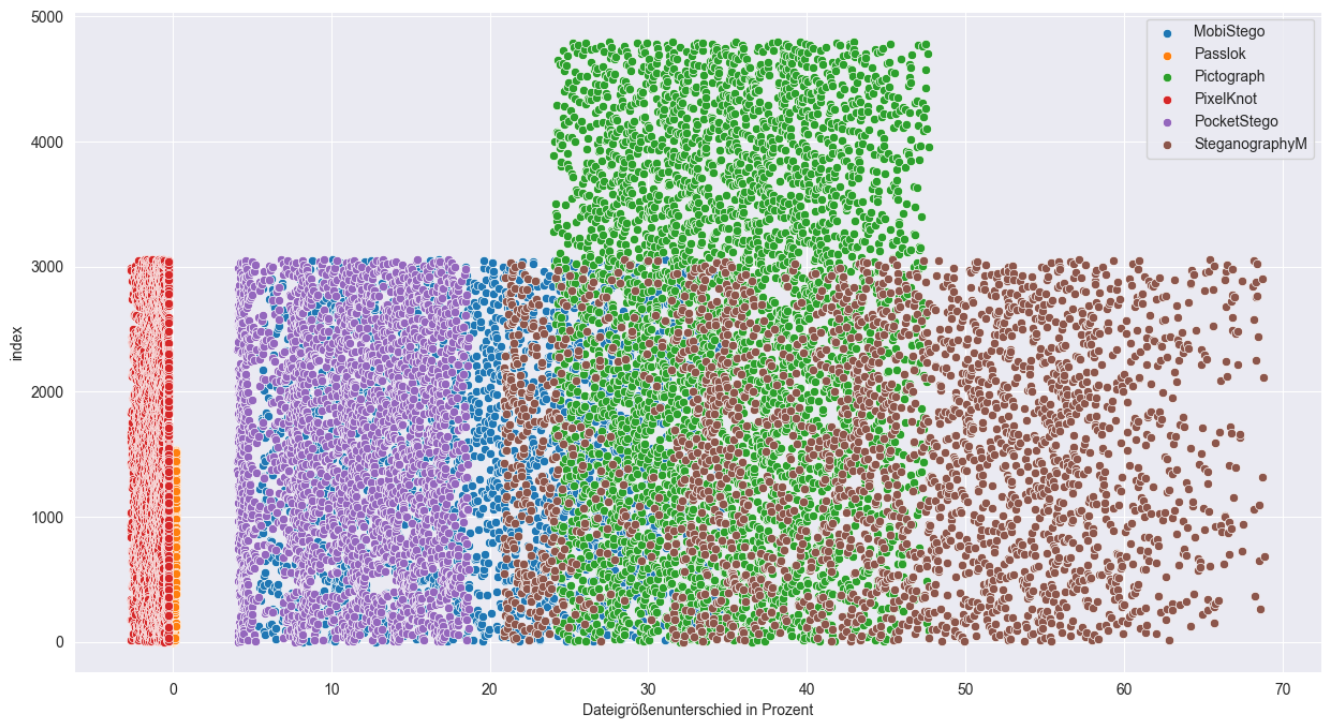


Abbildung 11. Darstellung der Dateigrößenunterschiede ohne Ausreißer

Ganz links ist PixelKnot in Rot neben Passlok in Orange im Bereich der negativen Dateigrößendifferenzen zu sehen. Nach einer Kluft folgen PocketStego in Lila im positiven Bereich mit starken Überlappungen mit MobiStego in Blau. MobiStego überschneidet sich zudem mit SteganographyM in Hellorange und Pictograph in Grün.

Die Auswertung der konkreten Werte ergeben die folgenden Wertebereiche:

Wertebereiche und abgeleitete Merkmale der Dateigrößenunterschiede

1. -2,63 % bis -0,25 % \Rightarrow Hinweis auf PixelKnot
2. 0,02 % bis 0,23 % \Rightarrow Hinweis auf Passlok
3. 4,09 % bis 5,44 % \Rightarrow Hinweis auf PocketStego
4. 5,44 % bis 18,68 % \Rightarrow Hinweis auf PocketStego oder MobiStego
5. 18,68 % bis 20,88 % \Rightarrow Hinweis auf MobiStego
6. 20,88 % bis 24 % \Rightarrow Hinweis auf MobiStego oder SteganographyM
7. 24 % bis 34,6 % \Rightarrow Hinweis auf MobiStego oder SteganographyM oder Pictograph
8. 34,6 % bis 47,67 % \Rightarrow Hinweis auf SteganographyM oder Pictograph
9. 47,67 % oder mehr \Rightarrow Hinweis auf Pictograph

Durch die fünf Intervalle 1, 2, 3, 5 und 9 lassen sich die betrachteten Stego-Apps basierend auf dem Datensatz eindeutig zu ordnen. In den anderen Fällen ist es lediglich einer von zwei beziehungsweise drei Apps zuordenbar. Diese Wertebereiche werden gleichzeitig als Merkmale angesehen und wurden entsprechend als Regeln umgesetzt.

4.4.3. LSB-Extraktion

Mittels der [LSB-Extraktion](#) wurde insbesondere nach Signaturen in der eingebetteten Nachricht gesucht. Um dies zu erreichen, wurden zunächst die genauen Einbettungsstrategien ermittelt. Dies erfolgte durch Probieren möglicher Konfigurationen mit den ersten 10 Bilderpaaren, auf die eine Stego-App angewendet wurde, und anschließend Validieren über alle Bildpaare der zugehörigen App. Eine Konfiguration besteht aus:

Einbettungsmethode *method*

Die verwendete Stego-App zum Einbetten

Optionen: Die Namen der Stego-Apps als String

Farbkanäle *channels*

Die zu überprüfenden Farbkanäle

Optionen: Jegliche Kombination aus **R**, **G**, **B**, **A**

LSBs *bits*

Die zu extrahierenden Least Significant Bits pro gewählten Farbkanal

Optionen: 1, 2, 4

Bit-Reihenfolge *endian*

Die bei der Zusammensetzung der Nachricht zu beachtende Bit-Reihenfolge

Optionen: **little** für Little-Endian oder **big** für Big-Endian

Iterationsrichtung *direction*

Die Richtung, in der über die Pixel iteriert werden soll

Optionen: **row** für die Iteration über die Zeile bzw. Bildweite oder **col[umn]** für die Iteration über die Spalte bzw. Bildhöhe

Der Ablauf wurde wie folgt umgesetzt:

1. Erhalte Konfiguration
2. Für jedes Stego-Bild
 - a. Suche und lese Originalnachricht aus
 - b. Extrahiere eingebettete Nachricht aus LSBs der Konfiguration folgend
 - c. Suche in extrahierter Nachricht nach Original
 - Wenn Suche erfolgreich war, füge Konfiguration und Index der gefundenen Nachricht der Ergebnisliste hinzu
3. Entferne Duplikate aus Ergebnisliste und gib zurück:
 - Wenn Länge der Ergebnisliste 1 ist, gib Erfolgsrate und Konfiguration zurück
 - Wenn Länge der Ergebnisliste größer 1 ist, gib Erfolgsrate und Ergebnisliste mit

passenden Konfigurationen zurück

- Ansonsten, gib Erfolgsrate und **None** zurück

Das Ergebnis wird nachfolgend in einer Tabelle dargestellt.

Tabelle 8. Konfigurationen für die LSB-Extraktion

Rate	Stego-App	Kanäle	Bits	Endianness	Richtung
1.0	MobiStego	RGB	2	big	zeilenweise
1.0	PocketStego	B	1	big	spaltenweise

MobiStego verwendet alle drei RGB-Kanäle bei 2-LSBs in zeilenweiser (oder Weite-orientierter) Iterationsrichtung mit eingebetteter Nachricht in Big-Endianness. PocketStego nutzt nur den B-Kanal beim LSB in spaltenweiser (oder Höhe-orientierter) Iterationsrichtung mit eingebetteter Nachricht in Big-Endianness. Dies deckt sich mit den Erkenntnissen in [33]. Da PocketStego und MobiStego die einzigen beiden Stego-Apps sind, die einen nicht-zufälligen LSB-Einbettungspfad verwenden, konnten auch nur diese erkannt werden. Steganography_M dagegen nutzt einen pseudo-randomisierten Einbettungspfad und Pictograph und PixelKnot setzen Steganografie in der [Frequenz-Domäne](#) ein und sind daher so nicht detektierbar.

Durch Vergleichen der extrahierten Nachrichten basierend auf den erkannten Konfigurationen konnten dann die folgenden Signaturen, wie auch in [33] beschrieben, entdeckt werden:

- **MobiStego:**
 - **@!#** (ASCII-Zeichen): Markiert den Beginn der Nachricht
 - **#!@** (ASCII-Zeichen): Markiert das Ende der Nachricht
- **PocketStego:**
 - **b'\x00'** (Byte): Markiert das Ende der Nachricht. Es handelt sich dabei um den Null-Byte.

Somit lassen sich folgende Merkmale definieren:

Abgeleitete Merkmale der LSB-Extraktion

Signaturen **@!#** und **#!@**

- Wenn die Signaturen **@!#** und **#!@** bei der Extraktion der 2-LSB-Ebene spaltenweise aus den RGB-Kanälen der Pixel, und im Big-Endian-Format zusammengesetzt, gefunden werden → Hinweis auf MobiStego

Signatur **b'\x00'**

- Wenn die Signatur **b'\x00'** bei der Extraktion der 1-LSB-Ebene zeilenweise aus dem B-Kanal der Pixel, und im Big-Endian-Format zusammengesetzt, gefunden wird → Hinweis auf PocketStego

4.5. Konfiguration des Detektors

Aus den vorherigen Ergebnissen lassen sich die Regeln ableiten, welche der Detektor zur Analyse nutzen wird. Die dazugehörige Datei ist im [Anhang](#) beigefügt.

Die Regeldatei listet zunächst alle Tools auf, die detektiert werden können. Dann werden die Regeln definiert. Der daraus resultierende Regelbaum wird nachfolgend dargestellt.

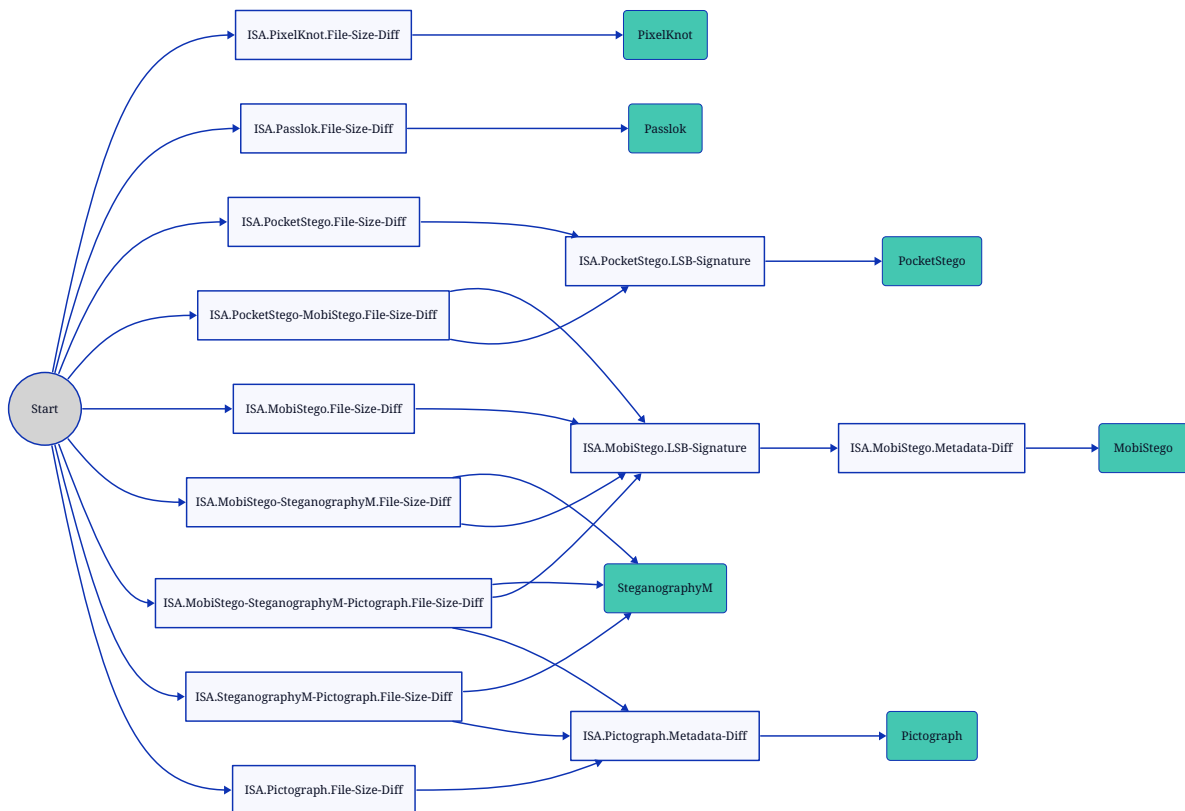


Abbildung 12. Regelbaum in Konfiguration des Detektors

Im ersten Regelblock werden die [Merkmale der Dateigrößenunterschiede](#) als Regeln `ISA.MobiStego.File-Size-Diff`, `ISA.PocketStego.File-Size-Diff`, `ISA.PocketStego-MobiStego.File-Size-Diff`, `ISA.MobiStego-SteganographyM.File-Size-Diff`, `ISA.MobiStego-SteganographyM-Pictograph.File-Size-Diff`, `ISA.SteganographyM-Pictograph.File-Size-Diff` und `ISA.Pictograph.File-Size-Diff` abgebildet. Daraufhin wird die [LSB-Signatur](#) von MobiStego durch `ISA.MobiStego.LSB-Signature` und respektive PocketStego durch `ISA.PocketStego.LSB-Signature` überprüft. Im Fall von MobiStego folgt darauf die Überprüfung der [Änderung von PNG:SignificantBits](#) sowie der [Änderung von PNG:ColorType](#) durch `ISA.MobiStego.Metadata-Diff`. Liefert dagegen die Analyse der Dateigrößenunterschiede zu Beginn einen Hinweis auf Pictograph, wird das Merkmal [Änderung von PNG:ColorType](#) durch `ISA.Pictograph.Metadata-Diff` vorgenommen.

Die Namen der Regeln bestehen immer aus dem Kürzel **ISA** für *Image Stego App*, gefolgt von der oder den spezifischen App(s) und dem überprüften Merkmal abgekürzt in Englisch. Dadurch werden Namensräume für die Merkmale mit Zuordnung zu den Stego-Apps geschaffen. Die Benennung ist aber funktional nur als Text abgebildet und könnte beliebig anderweitig

vorgenommen werden.

Die Regeln sind im Rahmen dieser Arbeit alle gleich gewichtet, da eine statistische Auswertung jeder einzelnen Regel nötig ist. Dies ist aus Gründen der Übersichtlichkeit zukünftigen Arbeiten überlassen. Ein sinnvoller Ansatz dazu wird im [Ausblick](#) beschrieben.

4.6. Auswertung des Detektors

Mit den zuvor vorgestellten Regeln wurde jedes der 18570 Cover-Stego-Bildpaare durchlaufen, um die verwendete Stego-App zu bestimmen. Die Ergebnisse wurden in Wahrheitsmatrizen dargestellt. Dabei ist immer links oben die Anzahl der wahr-positiven (TP), rechts oben der falsch-positiven (FP), links unten der falsch-negativen (FN) und rechts unten der wahr-falschen (TN) Detektionen wiederzufinden.

4.6.1. Auswertung mit binärer Betrachtung

Es wurde zunächst überprüft, ob eine Erkennung stattfand oder nicht. Jede Detektion mit einer Gewichtung größer als 0 wurde somit als positiv oder andernfalls als negativ eingestuft.

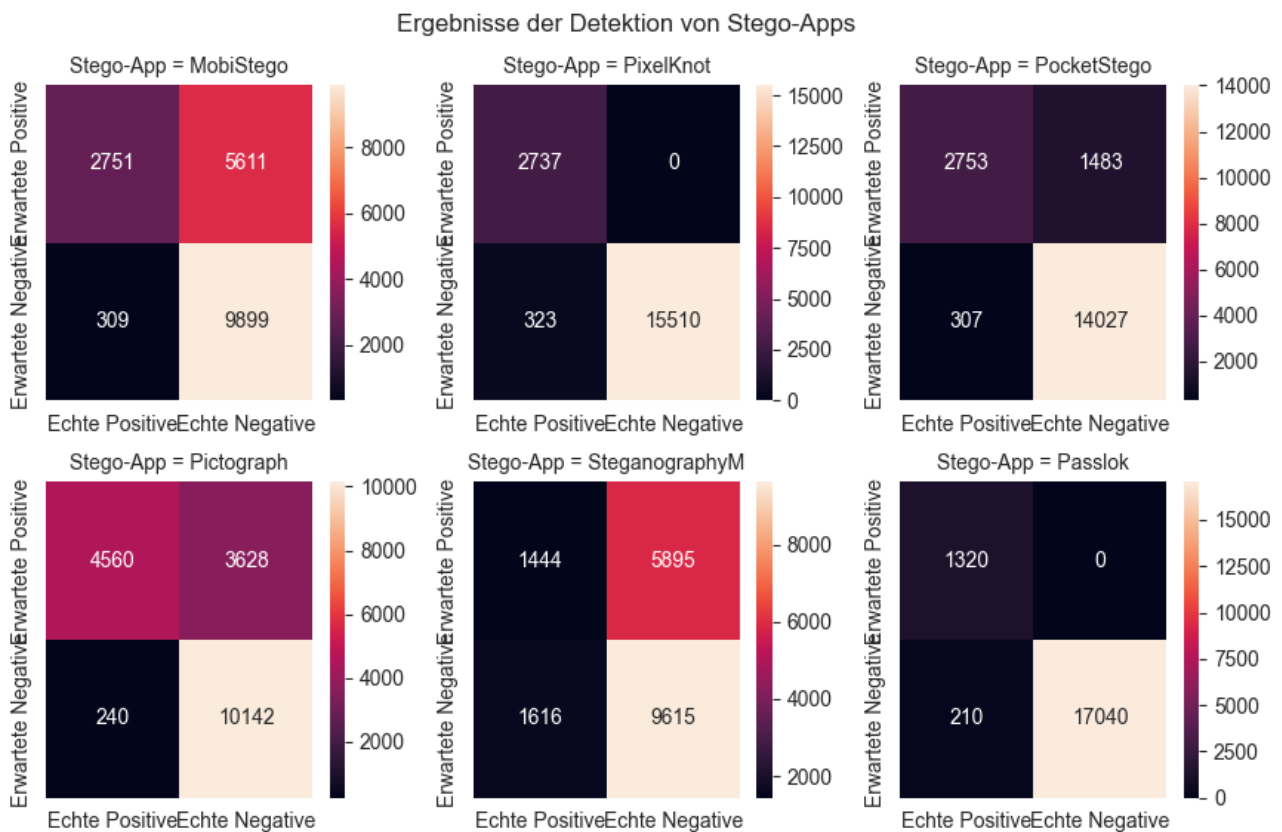


Abbildung 13. Ergebnisse der Detektion von Stego-Apps unter binärer Betrachtung

Es zeigt sich, dass mehr Stego-Bilder falsch MobiStego zugeordnet wurden als richtige. Die Rate der korrekt erkannten Negativen ist dagegen mit 9899 zu 309 sehr gut. Im Fall von PixelKnot wurde beinahe 1500 Bilder falsch dieser App zugeordnet. PocketStego weist sehr ähnliche Ergebnisse wie PixelKnot auf. Für Pictograph wurden beinahe ebenso viele falsch wie richtig zugeordnet. SteganographyM schließt am schlechtesten ab. Mehr als doppelt so viele Bilder werden falsch der App zugeordnet als richtigerweise. Ebenso ist die Anzahl an Falschnegativen im Vergleich zu allen anderen Apps sehr hoch. Passlok wurden die meisten Bilder passend zugeteilt oder als andere Stego-App identifiziert. Bemerkenswert ist, dass es dabei keine falsche Negativ-Zuordnung gab. Insgesamt lässt sich feststellen, dass die Rate der Falschzuordnung bei allen Stego-Apps außer SteganographyM sehr niedrig ist, was sich auch in den nachfolgenden Metriken widerspiegelt.

Die folgende Tabelle stellt die Ergebnisse mit den Metriken der Genauigkeit (*Accuracy*), Präzision (*Precision*) und Sensitivität (*Recall*) wie in [Abschnitt 2.2.1](#) beschrieben in Prozent dar.

Tabelle 9. Metriken der Detektion von Stego-Apps unter binärer Betrachtung

Stego-App	Genauigkeit	Präzision	Sensitivität
MobiStego	68,12	32,90	89,90
PixelKnot	98,26	100,00	89,44
PocketStego	90,36	64,99	89,97
Pictograph	79,17	55,69	95,00
SteganographyM	59,55	19,68	47,19
Passlok	98,87	100,00	86,27

Im Durchschnitt ergeben sich die folgenden Werte:

Tabelle 10. Durchschnittswerte der Metriken der Detektion von Stego-Apps unter binärer Betrachtung

Genauigkeit	Präzision	Sensitivität
82,38	62,20	82,96

Unter der Bedingung, dass eine Detektion immer gewertet wird, weisen SteganographyM und MobiStego mit 59,55 % und 68,12 % die schlechteste Genauigkeit auf, während Passlok, PixelKnot und PocketStego mit 98,86 % und 91,18 % und 90,36 % am besten abschneiden.

Die Präzision ist bei allen Stego-Apps mit Ausnahme von Passlok und PixelKnot relativ niedrig, sodass eine positive Erkennung nur mit einer Wahrscheinlichkeit zwischen ca. 20 bis 65 % richtig ist. Bei Passlok und PixelKnot dagegen wird jedes Bild korrekt der App zugeordnet, während bei SteganographyM mit 19,67 % nur jedes fünfte Bild richtigerweise der App zugewiesen wird.

Die Sensitivität ist bei allen Apps bis auf SteganographyM um die 90 % hoch. Es werden also die meisten Bilder, die von der jeweiligen App erstellt wurden, auch erkannt.

4.6.2. Auswertung unter Bezugnahme von Gewichtung

In der zweiten Auswertung wurden ausschließlich Detektionen als positiv eingestuft, wenn sie unter allen Detektionen für ein bestimmtes Cover-Stego-Paar die höchste Gewichtung vorweisen konnten. Bei gleicher Gewichtung wurde die Detektion aufgrund mangelnder Eindeutigkeit als negativ eingestuft.

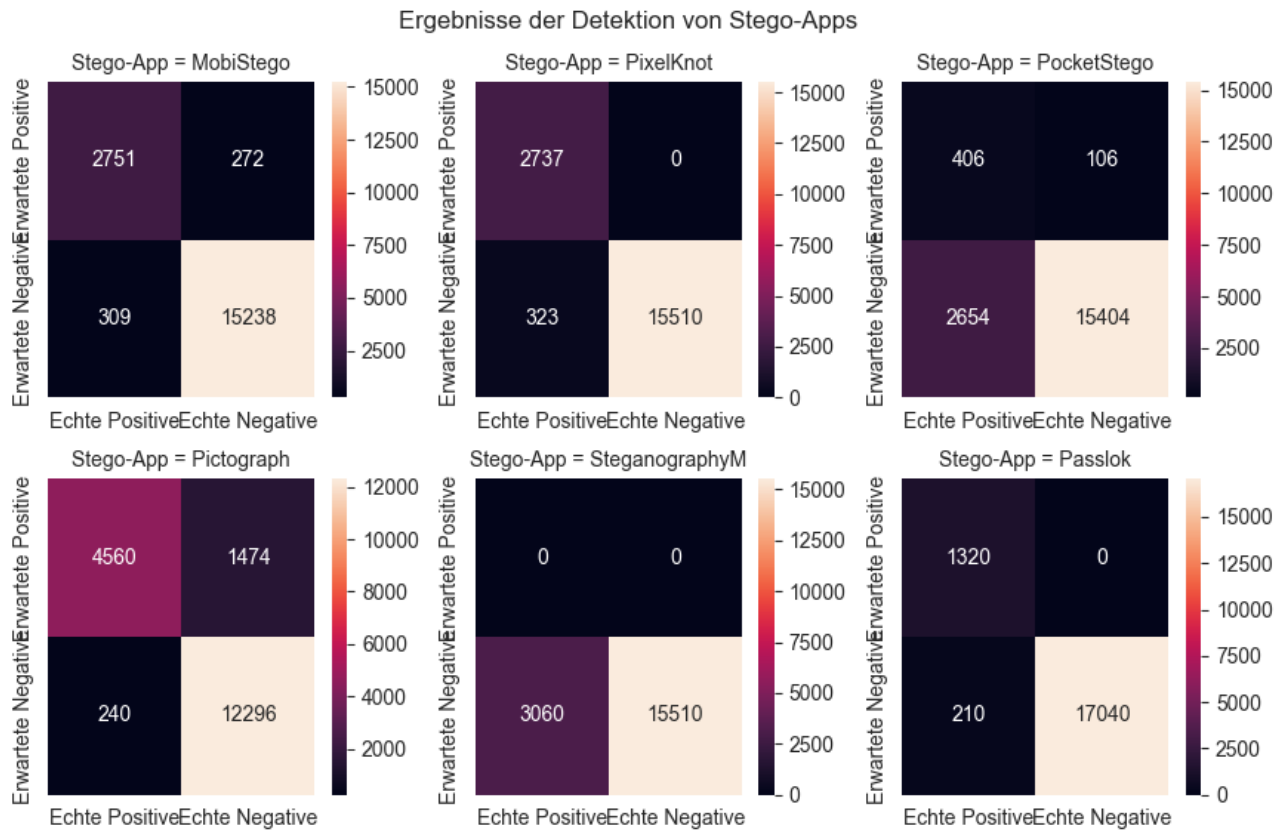


Abbildung 14. Ergebnisse der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

Unter Bezugnahme der Gewichtung stellt sich heraus, dass nun mehr Stego-Bilder richtig MobiStego zugeordnet wurden. Ebenso wurde bei PixelKnot weitaus seltener falsch zugeordnet, als es bei der vorherigen Auswertung geschehen ist. Dagegen schließt PocketStego deutlich schlechter ab, da die meisten echten Positivzuordnungen jetzt als Negative gewertet werden. Pictograph profitiert bei dieser Betrachtungsweise mit ca. 2000 weniger falschen Positivzuordnungen. SteganographyM schließt wieder am schlechtesten ab. Es wurden diesmal keine positiven Detektionen vermerkt, weshalb die Anzahl der Falsch-Negativen genau der Anzahl der Cover-Stego-Bildpaare der App entspricht. Somit ist die Rate der Falschzuordnung bei den Stego-Apps PocketStego und SteganographyM schlechter.

Die folgenden Werte zeigen diese Zusammenhänge wie bei [Tabelle 9](#) in Prozent.

Tabelle 11. Metriken der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

Stego-App	Genauigkeit	Präzision	Sensitivität
MobiStego	96,87	91,00	89,90
PixelKnot	98,26	100,00	89,44
PocketStego	85,14	79,30	13,27

Stego-App	Genauigkeit	Präzision	Sensitivität
Pictograph	90,77	75,57	95,00
SteganographyM	83,52	0,00	0,00
Passlok	98,87	100,00	86,27

Im Durchschnitt wurden die folgenden Werte erzielt:

Tabelle 12. Durchschnittswerte der Metriken der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

Genauigkeit	Präzision	Sensitivität
92,23	74,31	62,31

Wenn ausschließlich Detektionen mit der höchsten Gewichtung als positiv gewertet werden, zeigen sich bei allen Apps sehr gute Genauigkeitswerte mit 83,52 % bei SteganographyM als schlechtes und 98,29 % bei PixelKnot als bestes Ergebnis. Dies scheint auf eine merkbare Verbesserung im Vergleich zur [Auswertung mit binärer Betrachtung](#) hinzudeuten.

Bei der Präzision zeigt sich zunächst ein ebenfalls besseres Bild. Insbesondere MobiStego schneidet mit einer Verbesserung um fast 60 % deutlich besser ab. Auch PocketStego und Pictograph konnten 15 bis 30 % zulegen. Die Werte von SteganographyM dagegen liegen bei 0 %. PixelKnot und Passlok verzeichnen erneut eine Präzision von 100 %.

Die Sensitivität blieb bei MobiStego, PixelKnot und Pictograph im Vergleich zur [Auswertung mit binärer Betrachtung](#) unverändert. Bei PocketStego sank diese um mehr als 70 % ab, während sich die Werte von SteganographyM wieder bei 0 % befinden.

4.6.3. Fazit zu den zwei Auswertungen

Es zeigt sich ein relevanter Unterschied in der Betrachtungsweise. Bei der [Auswertung unter binärer Betrachtung](#) ist die Sensitivität mit 82,96 % um 20 % höher als im Fall der [Auswertung unter Bezugnahme von Gewichtung](#). Dagegen ist sowohl die Genauigkeit als auch die Präzision bei letzterem um mehr als 10 % höher. Eine mögliche Erklärung ist der Aufbau des Regelbaums. Da zunächst die Dateigrößenunterschiede verglichen werden, kommt es hier frühzeitig zur Erkennung mehrerer Stego-Apps, die bei der Betrachtung mit höchster Gewichtung jedoch ignoriert werden.

Im Allgemeinen ist die [Auswertung unter Bezugnahme von Gewichtung](#) von beiden vorzuziehen, da sie im Durchschnitt bessere Ergebnisse erzielt, wenn die Korrektheit der Ergebnisse sehr wichtig ist. Geht es dagegen darum, alle möglichen Stego-Apps zu detektieren, sollte auf die [Auswertung unter binärer Betrachtung](#) zurückgegriffen werden.

Abschließend ist zu erwähnen, dass durch eine richtige Gewichtung der möglichen Tools in jeder Regel noch bessere Ergebnisse möglich werden könnten. Ideen werden dazu im [Abschnitt 6.3](#) vorgestellt. Außerdem könnte eine Umstrukturierung des Regelbaums Verbesserungen bringen, indem beispielsweise erst die Metadaten untersucht werden.

5. Schlussfolgerung

Für diese Arbeit wurden [vier Ziele](#) formuliert, welche als **Z.1** bis **Z.4** definiert wurden.

Für **Z.1** sollte eine Liste aus Malware-Vorkommnissen erstellt werden, in denen Information Hiding Techniken wie Steganografie verwendet wurde, um die Relevanz des Themas sowie die konkrete Verwendung außerhalb des akademischen Umfelds zu untersuchen. Dies wurde im [Abschnitt 4.1](#) durchgeführt. Es zeigten sich einige Fälle mit Struktureinbettungsverfahren, bei denen beispielsweise an das Ende der Bilddatei angehängen oder in die Metadaten eingebettet wurde. Am meisten wurden LSB- und XOR-Verfahren zur Einbettung in die Raum-Domäne verzeichnet, deren Fallzahl in der jüngeren Zeit zunahm. Die Frequenz-Domäne wurde durch Malware nicht ein Mal verwendet.

Aus den gewonnenen Erkenntnissen wurde für **Z.2** (Erstellung einer Auswahlliste an Stego- und Wasserzeichen-Software) in [Abschnitt 4.2](#) eine List von Information Hiding Tools zusammengestellt, die auf unterschiedlichen Plattformen wie Windows oder macOS sowie Android und iOS lauffähig sind. Außerdem wurden Tools zur Erstellung von Wasserzeichen untersucht. Es zeigte sich, dass Wasserzeichen hauptsächlich in der Frequenz-Domäne arbeiten, da ihr Fokus auf eine hohe Robustheit im Kontrast zu einer hohen Kapazität steht. Stego-Tools und -Apps dagegen sind verstärkt in der räumlichen Domäne aktiv, was den Trend von Malware widerspiegelt.

Bei der Untersuchung der Tools stellte sich heraus, dass für die Studie von Stego-Apps eine umfangreiche Bilddatenbank mit Cover-Stego-Bildpaaren erstellt wurde und diese öffentlich zugänglich ist. Somit war die Nutzung dieser zum Erreichen von **Z.3** (Recherche und Aufbau von Forschungsdaten) naheliegend.

Schließlich wurde zur Erfüllung von **Z.4** (Entwicklung eines Detektor-Prototyps) anhand der Forschungsdaten Möglichkeiten untersucht, die Stego-Apps nur mittels der Cover- und Stego-Bilder zu erkennen. Dazu wurden zum einen die Metadaten untersucht. Es zeigte sich, dass zwei Apps PNG-Metadaten änderten. Zum anderen wurden die Differenzen in den Dateigrößen der Stego- zu ihren Cover-Bildern analysiert. Daraus konnten Intervalle bei den prozentualen Unterschieden identifiziert werden, die Hinweise auf bestimmte Stego-Apps lieferten. Schließlich konnte bei zwei Stego-Apps, die mittels LSB-Verfahren in die Raum-Domäne einbetten, Signaturen bei der Extraktion der Least Significant Bits festgestellt werden, welche im Fall von MobiStego ein eindeutiges Fingerprinting der App zulassen.

Nach Abarbeitung der Ziele können nun die [Forschungsfragen F.1 bis F.3](#) beantwortet werden.

Es bestätigt sich die Hypothese zu **F.1**, wobei lediglich im Fall von MobiStego mit der Signatur auf der LSB-Ebene ein voll belastbares Merkmal gefunden werden konnte. In allen anderen Fällen konnte lediglich aus dem Zusammenspiel mehrerer Merkmale auf die mögliche Verwendung einer bestimmten Stego-App geschlossen werden.

Für **F.2** konnte eine Schnittmenge an Merkmalen zwischen Malware und Steganografie-Tools entdeckt werden, welche insbesondere durch die Verwendung von LSB-Verfahren gebildet werden. Nach aktuellem Stand sind Wasserzeichen-Tools im Frequenz-Bereich oder mittels neuronalen Netzen aktiv, während Malware noch keine derartigen Steganografie-Techniken nutzen und es

somit keine Gemeinsamkeiten von Malware und Wasserzeichen-Tools diesbezüglich gibt.

Aus diesen Erkenntnissen zeigt sich, dass die Hypothese zu **F.3** eingetreten ist. Die meisten böswilligen Programme sind auf Performance und Kapazität bei mäßiger Detektierbarkeit fokussiert, während Wasserzeichen robust gegen Angriffe auf ihr Verfahren sein müssen. Mit LSB-Verfahren sind einige Malware-Vorkommnisse schwerer detektierbar, allerdings bei geringerer Kapazität. Dieser Ausgleich wird auch bei den meisten Stego-Tools in Kauf genommen.

6. Ausblick

Der folgende Ausblick geht zunächst auf allgemeine Anknüpfungspunkte zu dieser Arbeit ein. Danach werden in weiteren Abschnitten Ausbau- und Verbesserungsmöglichkeiten des Forschungsdatensatzes und des Detektors vorgestellt.

6.1. Allgemein

In dieser Arbeit wurden drei grundlegende Merkmalsräume untersucht. Diese wurden spezifisch für die untersuchten Stego-Apps ausgewählt und führten auch zu spezifischen Ergebnissen. Allerdings sind nicht nur die zu untersuchenden Stego-Tools relevant für die Merkmalsuche, sondern auch der verwendete Datensatz. Der gewählte Datensatz wies keine Metadaten oder sonstige spezifische Eigenschaften von bestimmten Kameras auf, die zum Beispiel Rückschlüsse auf den Benutzer des Stego-Tools oder eine Lokalisation der Bilderstellung zugelassen hätten. Mit derartigen Daten, die ein Kamera-Fingerprinting ermöglichen, könnte neben dem verwendeten Tool auch Erkenntnisse über den Anwender des Tools gesammelt werden. Eine weitere Möglichkeit dazu ist die Verwendung von künstlicher Intelligenz zur Erkennung von Sprache in extrahierten Payloads. So hätte im Datensatz dieser Arbeit festgestellt werden können, dass englische Texte eingebettet wurden. Mit den modernen großen Sprachmodellen (LLMs) könnte versucht werden, sogar zu erkennen, dass es sich um Werke von Shakespeare handelt. In diesem Beispiel würde dies darauf hindeuten, dass der Anwender des Tools ein Liebhaber klassischer englischer Literatur ist und somit womöglich aus dem englischsprachigen Raum stammt oder sich dort schon lange aufhält. Bei Payloads aus Malware könnte dagegen die verwendete Programmiersprache und möglicherweise besondere Zeichen aus nicht-lateinischen Alphabeten erkannt werden, die ebenfalls auf die Herkunft der Entwickler der Malware schließen lassen könnten. Bei einer solchen Herangehensweise ist jedoch zu bedenken, dass zukünftige Abwehrmechanismen gegen diese Erkennungen bewusst falsche Fährten legen könnten.

6.2. Forschungsdatensatz

Der Forschungsdatensatz ließe sich vielfältig erweitern, um weitere Untersuchungen zu ermöglichen. Die folgende Liste stellt einige mögliche Erweiterungspunkte vor.

Liste möglicher Erweiterungen des Forschungssatzes

1. **Diversere Einbettungsnachrichten:** Zum Beispiel wurden nur Texte aus dem Englischen eingebettet. Womöglich würden nicht-lateinische Sprachen jedoch zu erkennbaren Mustern auf der LSB-Ebene führen. Ebenso könnte eingebetteter Programmcode durch die Wiederholung von Schlüsselworten eine Mustererkennung erlauben oder die Dateigröße der Stego-Bilder berechenbar verändern.
2. **Diversere Stego-Tools:** Diversere Stego-Tools könnten zu weiteren übergreifenden Erkenntnissen im Forschungsgebiet des Information Hidings führen.
3. **Größere Bildvielfalt:** Die untersuchten Bildpaare waren in den Grauton konvertiert. Sowohl für die Stego-Verfahren in der Raum- als auch in der Frequenz-Domäne sind die Farbkanäle für die Einbettung von Belang. Farbige Bilder mit unterschiedlichen Werten im Alpha-Kanal könnten ebenfalls neue Merkmale liefern.

6.3. Detektor

Nachfolgend werden offenen Punkte des Detektors gelistet. Diese weiteren Features und Optimierungen könnten in einer weiteren Ausbaustufe umgesetzt sein.

Liste der offenen Punkte des Detektors

1. **Gewichtung der Regeln:** Im Rahmen dieser Arbeit wurde auf eine komplexe Gewichtung der Ergebnisse von Regeln verzichtet, da dazu jede einzelne Regel statistisch hätte ausgewertet und verglichen werden müssen. Dies sollte bei einem weiteren Ausbau vorgenommen werden, um bessere Ergebnisse bei der Erkennung der Stego-Tools zu erreichen. Eine Herangehensweise ist, jeder der vorgestellten Regeln einzeln mit dem Datensatz und den Metriken aus [Abschnitt 4.6](#) zu evaluieren und alle Ergebnisse miteinander zu vergleichen. Dann ist eine Gewichtungsskala zu wählen, wie zum Beispiel 1 bis Anzahl der Regeln n . Die erfolgreichste/n Regel/n erhält/erhalten die höchste Gewichtung und die anderen entsprechend ihres Rangs in der Ergebnisliste eine niedrigere.
2. **Mehrere Unterbindungen in einer Regel:** Der aktuelle Aufbau der Regelstruktur wurde so vorgenommen, dass eine Regel immer einen Fall prüft. In der Praxis stellte sich aber heraus, dass es durchaus sinnvoll sein könnte, in einer Regel mehrere Unterbindungen prüfen zu können und so auch eine Unterbindung einem Tool zuweisen zu können. Auf diese Weise würde sich die aktuelle Duplizieren der Regeln deutlich verringern lassen.
3. **Auslagern von Regeln:** Es könnte sich auch als sinnvoll erweisen, die Regeln in einem flachen `rules`-Block zu definieren und dann nur per ID in einem Regelbaum zu referenzieren. Dies würde ebenfalls die Duplizieren von Regeln deutlich verringern und zudem den Regelbaum deutlich einfacher erweiterbar machen.
4. **Verwendung von YARA:** Anstelle der Verbesserung des bestehenden Detektors könnte die Umsetzung der Regeln mittels YARA angestrebt werden. So würde eine Verknüpfung mit einem bestehenden Standard aufgebaut würde und es könnte in den Regeln auf Funktionalität von YARA zurückgriffen werden. Aus Gründen der Komplexität der Erweiterung des YARA-Scanners wurde auf die Umsetzung des Detektors mittels YARA verzichtet.
5. **Automatische Generierung von Regeln:** Sowohl die YAML-basierte aktuelle Variante als auch YARA-Regeln könnten automatisiert mittels Python generiert werden. Dadurch würden sich mögliche Funde von Merkmalen schneller testen lassen.

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] J. Fridrich, „Applications of data hiding in digital images“, in *ISSPA '99. Proceedings of the Fifth International Symposium on Signal Processing and its Applications (IEEE Cat. No.99EX359)*, 1999, Bd. 1, S. 9, doi: 10.1109/ISSPA.1999.818099.
- [11] K. Joshi und R. Yadav, „New approach toward data hiding using XOR for image steganography“, in *2016 Ninth International Conference on Contemporary Computing (IC3)*, Aug. 2016, S. 1–6, doi: 10.1109/IC3.2016.7880204.
- [12] 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>.
- [13] M. Hossin und S. M.N, „A Review on Evaluation Metrics for Data Classification Evaluations“, *International Journal of Data Mining & Knowledge Management Process*, Bd. 5, S. 01–11, März 2015, doi: 10.5121/ijdkp.2015.5201.
- [14] 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.
- [15] L, „lucacav/steg-in-the-wild“. Feb. 01, 2024, Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://github.com/lucacav/steg-in-the-wild>.

- [16] 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>.
- [17] 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/>.
- [18] „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.
- [19] „MITRE ATT&CK®“. Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/>.
- [20] „Malpedia (Fraunhofer FKIE)“. Zugegriffen: Feb. 22, 2024. [Online]. Verfügbar unter: <https://malpedia.caad.fkie.fraunhofer.de/>.
- [21] „The Selenium Browser Automation Project“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://www.selenium.dev/documentation/>.
- [22] „pandas documentation — pandas 2.2.2 documentation“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://pandas.pydata.org/docs/>.
- [23] „Requests: HTTP for Humans™ — Requests 2.32.0 documentation“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://requests.readthedocs.io/en/latest/>.
- [24] „Welcome to BibtexParser’s documentation! — BibtexParser latest documentation“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://bibtexparser.readthedocs.io/en/main/>.
- [25] „Jupyter AI — Jupyter AI documentation“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://jupyter-ai.readthedocs.io/en/latest/>.
- [26] „Software \textbar MITRE ATT&CK®“. Zugegriffen: Feb. 23, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/software/>.
- [27] „Sliver, Software S0633 \textbar MITRE ATT&CK®“. Zugegriffen: Feb. 23, 2024. [Online]. Verfügbar unter: <https://attack.mitre.org/software/S0633/>.
- [28] U. Pilania, 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.
- [29] U. Pilania, 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.
- [30] 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.

- [31] „DominicBreuker/stego-toolkit: Collection of steganography tools - helps with CTF challenges“. Zugegriffen: Feb. 29, 2024. [Online]. Verfügbar unter: <https://github.com/DominicBreuker/stego-toolkit>.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] B. An u. a., „Benchmarking the Robustness of Image Watermarks“, Nr. arXiv:2401.08573. arXiv, Jan. 22, 2024, doi: 10.48550/arXiv.2401.08573.
- [36] 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>.
- [37] 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.
- [38] „NumPy documentation — NumPy v1.26 Manual“. Zugegriffen: Mai 20, 2024. [Online]. Verfügbar unter: <https://numpy.org/doc/stable/>.
- [39] „YARA - The pattern matching swiss knife for malware researchers“. Zugegriffen: Apr. 13, 2024. [Online]. Verfügbar unter: <https://virustotal.github.io/yara/>.
- [40] „os — Miscellaneous operating system interfaces“. Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://docs.python.org/3/library/os.html>.
- [41] „pathlib — Object-oriented filesystem paths“. Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://docs.python.org/3/library/pathlib.html>.
- [42] „Image Module“. Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://pillow.readthedocs.io/en/stable/reference/reference/Image.html>.
- [43] „Malicious Python Package Hides Sliver C2 Framework in Fake Requests Library Logo“. Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://thehackernews.com/2024/05/malicious-python-package-hides-sliver.html>.
- [44] muhan, „■■■■■■■■ ■■ ■■■ ■■(HWP) ■■■■ : RedEyes(ScarCruft)“. Feb. 14, 2023, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://asec.ahnlab.com/ko/47622/>.
- [45] „Operation SMN: Axiom Threat Actor Group Report“. Novetta, Nov. 01, 2014, [Online]. Verfügbar unter: https://web.archive.org/web/20230211014413/https://www.novetta.com/wp-content/uploads/2014/11/Executive_Summary-Final_1.pdf.
- [46] D. N. and A. Rubinfeld, „Diavol - A New Ransomware Used By Wizard Spider? \textbar

- Fortinet“. Juli 01, 2021, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://www.fortinet.com/blog/threat-research/diabol-new-ransomware-used-by-wizard-spider>.
- [47] „ProLock Ransomware“. Seurtek, [Online]. Verfügbar unter: <https://seuretek.com/wp-content/uploads/2018/10/Seuretek-Advisory-ProLock-Ransomware.pdf>.
- [48] „ObliqueRAT returns with new campaign using hijacked websites“. März 02, 2021, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://blog.talosintelligence.com/obliquerat-new-campaign/>.
- [49] „BishopFox/sliver“. Bishop Fox, Mai 17, 2024, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://github.com/BishopFox/sliver>.
- [50] M. Faou, M. Tartare, und T. Dupuy, „OPERATION GHOST The Dukes aren’t back they never left“. ESET Research, Okt. 01, 2019, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: https://web-assets.esetstatic.com/wls/2019/10/ESET_Operation_Ghost_Dukes.pdf.
- [51] „Malicious Python Package Hides Sliver C2 Framework in Fake Requests Library Logo“. Zugegriffen: Mai 16, 2024. [Online]. Verfügbar unter: <https://thehackernews.com/2024/05/malicious-python-package-hides-sliver.html>.
- [52] izcoser, „izcoser/stegpy“. Mai 12, 2024, Zugegriffen: Mai 16, 2024. [Online]. Verfügbar unter: <https://github.com/izcoser/stegpy>.
- [53] D. Petrov, „DimitarPetrov/stegify“. Mai 13, 2024, Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://github.com/DimitarPetrov/stegify>.
- [54] R. David, „RobinDavid/LSB-Steganography“. Mai 12, 2024, Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://github.com/RobinDavid/LSB-Steganography>.
- [55] R. Gibson, „ragibson/Steganography“. Mai 14, 2024, Zugegriffen: Mai 15, 2024. [Online]. Verfügbar unter: <https://github.com/ragibson/Steganography>.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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.
- [60] 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.

[61] rosuH, „rosuH/EasyWatermark“. Mai 17, 2024, Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://github.com/rosuH/EasyWatermark>.

[62] „seaborn.violinplot — seaborn 0.13.2 documentation“. Zugegriffen: Mai 17, 2024. [Online]. Verfügbar unter: <https://seaborn.pydata.org/generated/seaborn.violinplot.html>.

Tabellenverzeichnis

[Tabelle 1.](#) Wahrheitsmatrix mit TP, FP, FN und TN

[Tabelle 2.](#) Verwendete Analyse-Tools zur Implementierung des Detektor-Prototyps

[Tabelle 3.](#) Ausgewählte Analyseverfahren zur Implementierung des Detektor-Prototyps

[Tabelle 4.](#) Auszug aus Malware-Daten

[Tabelle 5.](#) Übersicht von Stego-Tools als Desktop-Anwendungen

[Tabelle 6.](#) Übersicht von Stego-Tools als mobile Anwendungen

[Tabelle 7.](#) Vor- und Nachteile der Verwendung der StegoAppDB als Forschungsdatensatzes

[Tabelle 8.](#) Konfigurationen für die LSB-Extraktion

[Tabelle 9.](#) Metriken der Detektion von Stego-Apps unter binärer Betrachtung

[Tabelle 10.](#) Durchschnittswerte der Metriken der Detektion von Stego-Apps unter binärer Betrachtung

[Tabelle 11.](#) Metriken der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

[Tabelle 12.](#) Durchschnittswerte der Metriken der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

Abbildungsverzeichnis

Abbildung 1. Verteilung von Kapazität, Robustheit und Nicht-Detektierbarkeit [[fridrich_applications_1999](#)>10]

Abbildung 2. Software-Tabelle der MITRE-Datenbank [[noauthor_software_nodate](#)>26]

Abbildung 3. Beispiel einer Software auf der MITRE-Webseite [[noauthor_sliver_nodate](#)>27]

Abbildung 4. Beispiel einer *Techniques Used*-Tabelle [[noauthor_sliver_nodate](#)>27]

Abbildung 5. Anzahl der Malware-Vorkommnisse pro Betriebssysteme

Abbildung 6. Anzahl der Malware-Vorkommnisse pro Medientyp

Abbildung 7. Anzahl der Malware-Vorkommnisse pro Stego-Verfahren

Abbildung 8. Einstellung des Suchformulars der StegoAppDB für den verwendeten Forschungsdatensatz [[newman_stegoappdb_2019](#)>34]

Abbildung 9. Übersicht der Dateigrößenunterschiede als Streudiagramme

Abbildung 10. Darstellung der Dateigrößenunterschiede mit Ausreißern

Abbildung 11. Darstellung der Dateigrößenunterschiede ohne Ausreißer

Abbildung 12. Regelbaum in Konfiguration des Detektors

Abbildung 13. Ergebnisse der Detektion von Stego-Apps unter binärer Betrachtung

Abbildung 14. Ergebnisse der Detektion von Stego-Apps unter Bezugnahme von Gewichtung

Abbildung 15. Einstellung StegoAppDB nicht verwendet

Code-Ausschnittsverzeichnis

Code-Ausschnitt 1: Web-Scraper für Software-Tabelle der MITRE-Datenbank
Code-Ausschnitt 2: Web-Scraper für einzelne Software-Seiten der MITRE-Datenbank
Code-Ausschnitt 3: Web-Scraper für Malpedia-Datenbank
Code-Ausschnitt 4: Auslesen der Malpedia-Bibliografie-Datei
Code-Ausschnitt 5: Extraktion der Links von stego-in-the-wild
Code-Ausschnitt 6: Zusammenführung und Bereinigung der gesammelten Malware-Vorkommnisse
Code-Ausschnitt 7: Implementierung des Metadatenvergleichs
Code-Ausschnitt 8: Implementierung des Dateigrößenunterschieds
Code-Ausschnitt 9: Implementierung der LSB-Extraktion
Code-Ausschnitt 10: Beispiel-Regel von der YARA-Webseite
Code-Ausschnitt 11: Beispiel einer Konfigurationsdatei des Detektors
Code-Ausschnitt 12: Ausgabe der Kommandozeilenparameter des Detektors
Code-Ausschnitt 13: Evaluierung des Detektors unter binärer Betrachtung
Code-Ausschnitt 14: Evaluierung des Detektors mit Einbeziehung der Gewichtung
Code-Ausschnitt 15: Auszug aus `notebooks/detect-algo.ipynb`, Code-Zelle 6
Code-Ausschnitt 16: Abgeleitete Regeln

Anhang A: Sonstiges

Code-Ausschnitt 15: Auszug aus `notebooks/detect-algo.ipynb`, Code-Zelle 6

```
MESSAGE_DIR = Path('../datasets/StegoAppDB_stegos_20240309-030352/message_dictionary')

def find_nth_substring(haystack, needle, n):
    start = haystack.find(needle)
    while start >= 0 and n > 1:
        start = haystack.find(needle, start + len(needle))
        n -= 1
    return start

def get_original_message(stego_img):
    img_row = info_file[info_file['image_filename'] == stego_img.name]
    msg_name = img_row['message_dictionary'].values[0]
    starting_line_index = img_row['message_starting_index'].values[0]
    msg_len = img_row['message_length'].values[0]
    full_msg = (MESSAGE_DIR / msg_name).read_text()
    start_index = find_nth_substring(full_msg, '\n', starting_line_index - 1) + 1
    return full_msg[start_index:start_index + msg_len].encode('utf-8')

def get_embedded_message(stego_img, bits, endian, channels, direction):
    return attacks.lsb_extract(stego_img, bits, channels=channels, endian=endian, direction=direction).tobytes()

def detect_used_method_and_bits(stego_images, path_parts):
    method, channels, bits, endian, direction = path_parts
    results = []
    for stego_img in tqdm(stego_images, desc=f'Cycling through {method} {channels} {bits}-LSB {endian} {direction}'):
        original_msg = get_original_message(stego_img)
        extracted_msg = get_embedded_message(stego_img, bits, endian, channels, direction)
        index = extracted_msg.find(original_msg)
        if index != -1:
            results.append((method, channels, bits, endian, direction, index))

    rate = len(results) / len(stego_images)
    results = set(results)
    if len(results) == 1:
        return rate, results.pop()
    elif len(results) > 1:
        return rate, results
    else:
        return rate, None

#detected_used_method_and_bits = await detect_used_method_and_bits(
#    stego_images_by_method['PocketStego'][:10],
#    ('PocketStego', 'B', 1, 'MSB')
#)
detected_used_method_and_bits = [
    (rate, values)
    for rate, values in for_each_image(detect_used_method_and_bits, take=10)
]
```

Code-Ausschnitt 16: Abgeleitete Regeln

```
isd: "1"
```



```

tools:
- name: MobiStego
  tags: [ Android, LSB ]
- name: PixelKnot
  tags: [ Android, F5 ]
- name: Pictograph
  tags: [ iPhone, LSB ]
- name: SteganographyM
  tags: [ Android, LSB ]
- name: Passlok
  tags: [ Android, LSB ]
- name: PocketStego
  tags: [ Android, LSB ]
rules:
- name: ISA.PixelKnot.File-Size-Diff
  desc: Check if the file size difference is -2,63 % to -0,25%.
  tools:
    - name: PixelKnot
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: -2.63 <= value <= -0.25
###
- name: ISA.Passlok.File-Size-Diff
  desc: Check if the file size difference is 0,02 % to 0,23 %.
  tools:
    - name: Passlok
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 0.02 <= value <= 0.23
###
- name: ISA.PocketStego.File-Size-Diff
  desc: Check if the file size difference is 4,09 % bis 5,44 %.
  tools:
    - name: PocketStego
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 4.09 <= value < 5.44
  next:
    name: ISA.PocketStego.LSB-Signature
    desc: Check if data extracted from the LSBs contains a signature.
    tools:
      - name: PocketStego
        weight: 1
    match:
      value: attacks.lsb_extract(stego.path, bits=1, channels='B', endian='big', direction='col').tobytes()
      cond: value in b'\x00'
###
- name: ISA.PocketStego-MobiStego.File-Size-Diff
  desc: Check if the file size difference is 5,44 % to 18,68 %.
  tools:
    - name: PocketStego
      weight: 1
    - name: MobiStego
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 5.44 <= value < 18.68
  next:
    operator: any
    rules:
      - name: ISA.MobiStego.LSB-Signature
        desc: Check if data extracted from the LSBs contains a signature.
        tools:

```

```

- name: MobiStego
  weight: 1
match:
  value: attacks.lsb_extract(stego.path, bits=2, endian='big').tobytes()
  cond: value[:3] == b'@!#' and b'#!@' in value[3:]
next:
  name: ISA.MobiStego.Metadata-Diff
  desc: Check if the significant bits of the PNG file and the color type are modified.
  tools:
    - name: MobiStego
      weight: 1
  match:
    value: attacks.metadata_diff(cover.path, stego.path)
    cond: value['PNG:SignificantBits'][0] == '8 8 8' and value['PNG:SignificantBits'][1] == '8 8 8' and
value['PNG:ColorType'][0] == 2 and value['PNG:ColorType'][1] == 6
- name: ISA.PocketStego.LSB-Signature
  desc: Check if data extracted from the LSBs contains a signature.
  tools:
    - name: PocketStego
      weight: 1
  match:
    value: attacks.lsb_extract(stego.path, bits=1, channels='B', endian='big', direction='col').tobytes()
    cond: value in b'\x00'

###
- name: ISA.MobiStego.File-Size-Diff
  desc: Check if the file size difference is 18,68 % to 20,88 %.
  tools:
    - name: MobiStego
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 18.68 <= value < 20.88
  next:
    name: ISA.MobiStego.LSB-Signature
    desc: Check if data extracted from the LSBs contains a signature.
    tools:
      - name: MobiStego
        weight: 1
    match:
      value: attacks.lsb_extract(stego.path, bits=2, endian='big').tobytes()
      cond: value[:3] == b'@!#' and b'#!@' in value[3:]
    next:
      name: ISA.MobiStego.Metadata-Diff
      desc: Check if the significant bits of the PNG file and the color type are modified.
      tools:
        - name: MobiStego
          weight: 1
      match:
        value: attacks.metadata_diff(cover.path, stego.path)
        cond: value['PNG:SignificantBits'][0] == '8 8 8' and value['PNG:SignificantBits'][1] == '8 8 8' and
value['PNG:ColorType'][0] == 2 and value['PNG:ColorType'][1] == 6
###
- name: ISA.MobiStego-SteganographyM.File-Size-Diff
  desc: Check if the file size difference is 20,88 % to 24 %.
  tools:
    - name: MobiStego
      weight: 1
    - name: SteganographyM
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 20.88 <= value < 24
  next:
    name: ISA.MobiStego.LSB-Signature
    desc: Check if data extracted from the LSBs contains a signature.
    tools:

```

```

- name: MobiStego
  weight: 1
match:
  value: attacks.lsb_extract(stego.path, bits=2, endian='big').tobytes()
  cond: value[:3] == b'@!#' and b'#!@' in value[3:]
next:
  name: ISA.MobiStego.Metadata-Diff
  desc: Check if the significant bits of the PNG file and the color type are modified.
  tools:
    - name: MobiStego
      weight: 1
  match:
    value: attacks.metadata_diff(cover.path, stego.path)
    cond: value['PNG:SignificantBits'][0] == '8 8 8' and value['PNG:SignificantBits'][1] == '8 8 8' and
value['PNG:ColorType'][0] == 2 and value['PNG:ColorType'][1] == 6
####
- name: ISA.MobiStego-SteganographyM-Pictograph.File-Size-Diff
  desc: Check if the file size difference is 24 % to 34,6 %.
  tools:
    - name: MobiStego
      weight: 1
    - name: SteganographyM
      weight: 1
    - name: Pictograph
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 24 <= value < 34.6
  next:
    operator: any
    rules:
      - name: ISA.Pictograph.Metadata-Diff
        desc: Check if the color type is modified.
        tools:
          - name: Pictograph
            weight: 1
        match:
          value: attacks.metadata_diff(cover.path, stego.path)
          cond: value['PNG:ColorType'][0] == 0 and value['PNG:ColorType'][1] == 2
      - name: ISA.MobiStego.LSB-Signature
        desc: Check if data extracted from the LSBs contains a signature.
        tools:
          - name: MobiStego
            weight: 1
        match:
          value: attacks.lsb_extract(stego.path, bits=2, endian='big').tobytes()
          cond: value[:3] == b'@!#' and b'#!@' in value[3:]
        next:
          name: ISA.MobiStego.Metadata-Diff
          desc: Check if the significant bits of the PNG file and the color type are modified.
          tools:
            - name: MobiStego
              weight: 1
          match:
            value: attacks.metadata_diff(cover.path, stego.path)
            cond: value['PNG:SignificantBits'][0] == '8 8 8' and value['PNG:SignificantBits'][1] == '8 8 8' and
value['PNG:ColorType'][0] == 2 and value['PNG:ColorType'][1] == 6
####
- name: ISA.SteganographyM-Pictograph.File-Size-Diff
  desc: Check if the file size difference is in the range of 34,6 % to 47,67 %.
  tools:
    - name: Pictograph
      weight: 1
    - name: SteganographyM
      weight: 1
  match:

```

```

value: attacks.size_diff([(cover.path, stego.path)])[0][3]
cond: 34.6 <= value < 47.67
next:
  name: ISA.Pictograph.Metadata-Diff
  desc: Check if the color type is modified.
  tools:
    - name: Pictograph
      weight: 1
  match:
    value: attacks.metadata_diff(cover.path, stego.path)
    cond: value['PNG:ColorType'][0] == 0 and value['PNG:ColorType'][1] == 2
###
- name: ISA.Pictograph.File-Size-Diff
  desc: Check if the file size difference is of 47,67 % or more.
  tools:
    - name: Pictograph
      weight: 1
  match:
    value: attacks.size_diff([(cover.path, stego.path)])[0][3]
    cond: 47.67 <= value
next:
  name: ISA.Pictograph.Metadata-Diff
  desc: Check if the color type is modified.
  tools:
    - name: Pictograph
      weight: 1
  match:
    value: attacks.metadata_diff(cover.path, stego.path)
    cond: value['PNG:ColorType'][0] == 0 and value['PNG:ColorType'][1] == 2

```

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:

☒ Auto Exposure

ISO

10 - 7000

☒ Manual Exposure

ISO

10 - 7000

Exposure Time

1/11000 - 1/2

Number of Images:

244680

Estimated Download Size:

175.15 Gigabyte(s)

Download

Search Again

Abbildung 15. Einstellung StegoAppDB nicht verwendet