fishy – Ein Framework zur Umsetzung von Verstecktechniken in Dateisystemen

Adrian V. Kailus[1] · Christian Hecht[1] · Thomas Göbel[1,2]
Lorenz Liebler[1,2]

[1]Hochschule Darmstadt
{adrian.v.kailus | christian.hecht}@stud.h-da.de

[2]da/sec – Biometrics and Internet Security Research Group
{thomas.goebel | lorenz.liebler}@h-da.de

Zusammenfassung

Der Begriff Anti-Forensik bezeichnet im Allgemeinen verschiedene Techniken mit denen IT-forensische Ermittlungen erschwert oder gar verhindert werden können. Eine Unterkategorie ist die Nutzung von Verstecktechniken, um schützenswerte oder geheime Daten vor einem unerwünschten Zugriff zu verschleiern. Die Manipulation eines Dateisystems durch Ausnutzung dessen struktureller oder konzeptioneller Eigenschaften ist ein häufig erwähnter Anti-Forensik-Ansatz. Die theoretischen Ansätze werden oftmals von den Autoren in eigenen Tools realisiert. Zumeist ist deren Codebasis nicht zugänglich und somit ist eine Wartbarkeit sowie die Reproduzierbarkeit nicht immer garantiert. Die praktische und einheitliche Umsetzung von Verstecktechniken ist vorteilhaft für eine spätere Evaluation von forensischen Softwarekomponenten. In dieser Arbeit präsentieren wir *fishy*, ein Framework zur Umsetzung und Erforschung von Dateisystem-basierten Verstecktechniken. Das in Python implementierte Framework ermöglicht die Erforschung, Sammlung und Sicherstellung der Reproduzierbarkeit verschiedener Verstecktechniken. Neben der Beschreibung des eigentlichen Frameworks und dessen Verwendung präsentieren wir zudem verschiedene rudimentäre Verstecktechniken für unterschiedliche Dateisysteme und diskutieren mögliche zukünftige Erweiterungen unserer Arbeit.

1 Einleitung

Das Feld der Anti-Forensik hat in den letzten Jahren immer mehr an Bedeutung gewonnen. Generell stellt dieses Themengebiet einen wichtigen Teil der digitalen Forensik dar und die Erforschung verschiedener Techniken gewinnt immer mehr an Relevanz. Die Ansätze zum Verstecken von sensiblen Daten reichen dabei von Techniken der Verschlüsselung auf verschiedenen Abstraktionsebenen, der Steganographie, bis hin zu Netzwerk-basierter Verschleierung. Verschiedene Taxonomien wurden in den letzten Jahren präsentiert, welche bekannte Techniken kategorisieren und zusammenfassen [Gar07, WFM13, CBB16].

Ein bekannter Teilbereich der Anti-Forensik ist das Verschleiern von sensiblen Daten auf einem Datenträger selbst. Die erweiterte Taxonomie von Conlan et al. [CBB16] erlaubt die Unterscheidung zwischen Verstecktechniken, welche die Daten mithilfe des Datenträgers oder des

P. Schartner · N. Pohlmann (Hrsg.) · D•A•CH Security 2018 · syssec (2018) 91-104.

genutzten Dateisystems verstecken. Für die unterschiedlichen Verstecktechniken wurden verschiedenste Implementierungen und Tools veröffentlicht. Im Bereich der Dateisystem-basierten Verstecktechniken ist die Autorenschaft der Tools zumeist gekennzeichnet, allerdings ist deren Quellcode und die Pflege der Projekte oftmals nicht ersichtlich[1]. Zusätzlich ist ein Großteil der vorgestellten Tools auf die Anwendung der Verstecktechnik innerhalb eines spezifischen Dateisystems ausgerichtet.

Die Art der Bereitstellung verhindert nicht nur die Validierung, Reproduzierbarkeit und Weiterentwicklung der Ansätze selbst, sondern auch die etwaige Evaluation vorhandener Forensik-Suites. Die Wichtigkeit der Evaluation von Softwaresystemen zur digitalen Auswertung von Beweismitteln ist innerhalb der Community bekannt und offensichtlich. Die Evaluation dieser Tools kann dabei grob in zwei Kategorien unterschieden werden: die Gewährleistung der Richtigkeit der Softwareprodukte [LWA08] sowie deren Sicherheit[WFM13, NPSB07, Rid09].

In dieser Arbeit stellen wir das in Python geschriebene Framework *fishy* (*Filesystem Hiding Techniques in Python*) vor. Mit Hilfe unserer Entwicklung ist es möglich Dateisystem-basierte Manipulationen zu implementieren, zu kategorisieren und zu testen. Durch den modularen Aufbau des Frameworks ermöglichen wir die Manipulation von mehreren Dateisystemen (aktuell NTFS, EXT4, FAT) und die zukünftige Integration weiterer Dateisysteme. Das Framework kann direkt über ein Command-Line Interface (CLI) genutzt werden oder als Bibliothek in ein bereits bestehendes Projekt eingebunden werden. Das Framework soll zukünftig IT-Forensikern und Wissenschaftlern einen Zugang zur Entwicklung und zum Testen von neuen Verstecktechniken geben. Bestehende und bereits diskutierte Ansätze können in dem Framework gesammelt und nachvollzogen werden. Ein weiteres zentrales Ziel, das wir mit der Entwicklung dieses Frameworks verfolgen, ist die Qualitätssicherung bestehender forensischer Verfahren und Softwaresuites. Unterschiedliche Ansätze und Verstecktechniken können nun, im Gegensatz zu vorherigen Entwicklungen, in einem zentralen Framework gesammelt und der IT-forensischen Community zur Verfügung gestellt werden. Damit wird auch zukünftig die Zugänglichkeit der Quellen garantiert. Das Framework wird als OpenSource-Projekt in Form eines GitHub Repository veröffentlicht[2]. Neben dem Framework selbst präsentieren wir einige beispielhafte Verstecktechniken, mit welchen die Nutzung des Frameworks und die mögliche Implementierung von Verstecktechniken demonstriert werden. Weitere technische Details und konkrete Anwendungsbeispiele finden sich im Repository des Projekts.

Zunächst stellen wir in Abschnitt 2 relevante Publikationen und konkrete Implementierungen vor, welche das Thema Anti-Forensik und speziell die Manipulation von Dateisystemen behandelt. In Abschnitt 3 erläutern wir den modularen Aufbau des Frameworks, die Nutzung und Kommandostruktur sowie mögliche Einschränkungen der aktuellen Implementierung. In Abschnitt 4 fassen wir die Ergebnisse unserer aktuellen Umsetzung zusammen und verweisen auf mögliche zukünftige Entwicklungen.

2 Literaturrecherche

In Abschnitt 2.1 werden wir zunächst kurz auf das weitläufige Themenfeld der Anti-Forensik eingehen und dabei den Fokus auf vorhandene Ansätze der Dateisystem-basierten Verschleierung lenken. Anschließend präsentieren wir in Abschnitt 2.2 eine kurze Übersicht über ver-

[1] http://www.forensicswiki.org/wiki/Anti-forensic_techniques\#Generic_Data_Hiding (letzter Zugriff 2018-06-10).

[2] https://github.com/dasec/fishy (letzter Zugriff 2018-06-10).

schiedene Tools, welche bereits in der Vergangenheit zum Verstecken von sensiblen Daten im Dateisystem genutzt wurden.

2.1 Anti-Forensik

Bekannte Definitionen zum Begriff Anti-Forensik stammen von Rogers [Rog05] und Harris [Har06]. Eine neuere Definition aus dem Jahr 2016 stammt von Conlan et al. [CBB16] und lautet wie folgt: „Any attempts to alter, disrupt, negate, or in any way interfere with scientifically valid forensic investigations“. Zur besseren Unterscheidung verschiedener Anti-Forensik Techniken wurde von Rogers [Rog05] eine Unterteilung in folgende vier Kategorien vorgenommen: *Data Hiding*, *Artifact Wiping*, *Trail Obfuscation* und *Attacks against the computer forensic process and tools*. Conlan et al. [CBB16] erweitern die in der forensischen Community anerkannte Taxonomie von Rogers um eine fünfte Kategorie: *Possible indications of anti-digital forensic activity*. Da mittlerweile eine Vielzahl an Anti-Forensik Techniken bekannt sind, unterteilen die Autoren die fünf Kategorien zusätzlich in mehrere Unterkategorien. Dies ermöglicht eine präzisere Zuordnung aktueller Anti-Forensik Techniken. Die im weiteren Verlauf der Arbeit genannten Verstecktechniken können der Kategorie *Data Hiding* bzw. der Unterkategorie *Filesystem Manipulation* der erweiterten Taxonomie zugeordnet werden.

Das Verstecken von Daten innerhalb von Dateisystemstrukturen wurde bereits 1998 von Anderson et al. [ANS98] im Zusammenhang mit der Entwicklung eines steganographischen Dateisystems behandelt. Daraus resultierte *StegFS*, ein steganographisches Dateisystem auf Basis von EXT2 [MK99]. In der Zwischenzeit sind viele weitere Methoden bekannt, die es ermöglichen in den verschiedenen vorhandenen Dateisystemen sensible Daten zu verstecken ohne die eigentliche Funktion des Dateisystems zu beeinträchtigen. Neben typischen Versteck-Bereichen wie File Slack, können auch Slack-Bereiche verschiedener Dateisystemstrukturen (z.B. Block- oder Inode-Bitmap eines EXT4-Dateisystems) zum Verstecken von Daten verwendet werden. Zudem können auch vorhandene Datenstrukturen (z.B. Zeitstempel) oder für zukünftige Dateisystem-Erweiterungen reservierte Bereiche (z.B. Reserved GDT Blocks) als Datenversteck missbraucht werden. In IT-forensischen Ermittlungen spielt die Kenntnis über diese auf den ersten Blick unauffälligen Bereiche durchaus eine entscheidende Rolle, beispielsweise zur Sicherung aller belastenden digitalen Spuren.

Folgende weitere Publikationen bieten relevante Anhaltspunkte für diese Arbeit, da sie allesamt unterschiedliche Verstecktechniken und mögliche Gegenmaßnahmen behandeln. Die folgende Auflistung vorhandener Publikationen zum Thema Verstecktechniken in Dateisystemen zeigt insbesondere welches Dateisystem bzw. welche Dateisystemversion im jeweiligen Paper behandelt wird: [EJ05] (EXT2/3, NTFS), [PDMS05] (EXT2/3), [Gru05] (EXT2), [HBW06] (NTFS), [KN07] (NTFS, EXT2/3), [BHS08] (EXT2/3, FAT, NTFS). Anhand unserer Auflistung wird deutlich, dass die Mehrzahl der Paper bereits veraltet ist und es wenige aktuelle Beiträge gibt. Zudem nehmen die meisten Paper immer nur Bezug auf ein bestimmtes Dateisystem. In einer weiteren Arbeit [Fai12] (EXT4) werden die wichtigsten Dateisystemstrukturen des EXT4 Dateisystems forensisch analysiert. Außerdem werden mehrere potentielle Verstecke in EXT4 erwähnt, diese werden allerdings nicht weiter fokussiert. Eine aktuelle Publikation [GB18a] (EXT4) untersucht verschiedene Anti-Forensik Techniken im EXT4 Dateisystem. Zusätzlich wird für jede Technik der verfügbare Speicherplatz und eine Einschätzung zum Schwierigkeitsgrad der Detektion versteckter Daten genannt.

2.2 Tool-Recherche

Während der Literaturrecherche wurden einige praktische Umsetzungen Dateisystem-basierter Verstecktechniken und zugehörige Dokumentation gefunden, wovon nun einige Beispiele folgen [Gar07, FL05]. Das Linux-basierte Tool `bmap`[3] ermöglicht das Verstecken von Daten innerhalb eines NTFS Slackbereichs. Es gibt keine offizielle Projektseite und kein vertrauenswürdiges Repository, in welchem das Tool zur Verfügung gestellt oder weiterentwickelt wird. Ein Windows-basiertes Tool namens `slacker`[4] ermöglicht ebenfalls das Verstecken von Daten innerhalb des NTFS Slackbereichs. Das Tool wurde von Bishop Fox entwickelt und zusammen mit anderen Anti-Forensik Tools (beispielsweise `Timestomp` – zur Manipulation von NTFS-Zeitstempel) unter dem Namen *Metasploit Anti-Forensic Investigation Arsenal* (kurz: *MAFIA*) in das Metasploit Framework integriert. Das Tool steht trotzdem aktuell nicht zum Download zur Verfügung. Das Tool `FragFS` [TM06] ist in der Lage Daten innerhalb der letzten 8 Bytes eines MFT-Eintrags im NTFS-Dateisystems zu verstecken. Zudem wurden bereits mehrere Anti-Forensik Tools für das ext-Dateisystem entwickelt. Diese sind zwischenzeitlich allerdings stark veraltet, da sie ursprünglich für EXT2 programmiert wurden [Gru05]. `Rune FS`[5] war beispielsweise in der Lage Daten unter Ausnutzung eines Fehlers vor der Forensik-Software The Coroner's Toolkit zu verbergen. Konkret wurden die Daten hier innerhalb der reservierten Bad Blocks Inode im ext-Dateisystem versteckt. `Waffen FS` fügt ein EXT3 Journal zu einem EXT2 Dateisystem hinzu, um anschließend beliebige Daten darin zu verstecken. `KY FS` manipuliert einen Verzeichniseintrags so, als würde der Eintrag nicht verwendet, um anschließend Daten im Verzeichniseintrag zu verstecken. `Data Mule FS` veranschaulicht, dass bereits in einem nur 1 GiB großen EXT2-Image unter Ausnutzung aller reservierten und durch Padding aufgefüllten Bereiche der Superblöcke, Gruppendeskriptoren und Inodes etwa 1 MiB freier Speicherplatz zum Verstecken von Daten übrig bleibt. `Timestamp-Magic` [GB18b] versteckt Daten innerhalb der Metadaten eines EXT4-Dateisystems. Genau genommen werden hierfür die Nanosekunden-Zeitstempel der Inode-Tabellen verwendet. `TOMS` [NVS+16] demonstriert ein ähnliches Verhalten, allerdings für das NTFS-Dateisystem.

Bei der Durchsicht bereits vorhandener Tools fällt auf, dass, bis auf wenige Ausnahmen, häufig der Quellcode zu den Tools nicht (mehr) öffentlich zugänglich ist. Zudem wurden viele der Tools ursprünglich für ältere Dateisystemversionen entwickelt, weshalb deren Codebasis nicht weiter gepflegt wird. *fishy* ermöglicht eine einheitliche Umsetzung von Verstecktechniken, sodass die Reproduzierbarkeit und Evaluation von Forensik-Software zukünftig erleichtert wird.

3 Fishy

Im Folgenden Abschnitt werden die prinzipielle Architektur und der modulare Aufbau des *fishy*-Frameworks erläutert. Neben der Installation und den dabei zu berücksichtigenden Abhängigkeiten wird auch die mögliche Erzeugung von Testimages beschrieben. Mit diesen Testimages lassen sich die einzelnen Funktionen und ihre Kommando-Struktur testen. Die Kommandos und deren Verwendung werden ebenfalls in den folgenden Sektionen beschrieben.

[3] http://www.gupiaoya.com/Soft/Soft_6823.htm (letzter Zugriff 2018-06-10).
[4] http://www.bishopfox.com/resources/tools/other-free-tools/mafia/ (letzter Zugriff 2018-06-10).
[5] http://index-of.es/Linux/R/runefs.tar.gz (letzter Zugriff 2018-06-10).

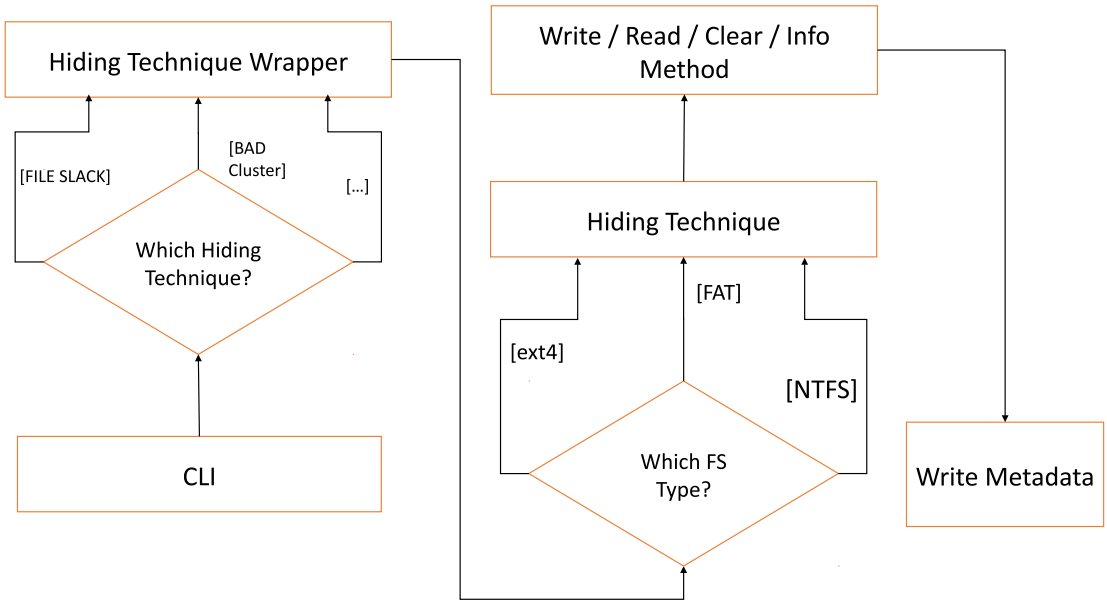

Abb. 1: Vereinfachte Übersicht der modularen Struktur von `fishy`.

3.1 Architektur

Beim Design des Frameworks wurde darauf geachtet die Architektur möglichst modular zu gestalten. Verschiedene Schichten ermöglichen das Kapseln von Funktionalitäten. Der logische Ablauf der Verarbeitung von Informationen durch das fishy Framework wird in Abbildung 1 dargestellt.

Zu Beginn wertet das Command-Line Interface (`CLI`) die eingegebenen Kommandozeilenparameter aus und ruft das entsprechende Kommando im `Hiding Technique Wrapper` auf, welcher die verwendete Verstecktechnik von anderen Techniken kapselt. Der `Hiding Technique Wrapper` überprüft das angegebene Dateisystem-Image nach dem Dateisystemtyp und ruft die entsprechende `Hiding Technique` Implementierung für dieses Dateisystem auf. Im Falle der Schreibmethode gibt die `Hiding Technique` Implementierung ein Metadaten-Objekt zurück, welches später benötigt wird um die Daten wiederherzustellen. Diese Metadaten werden in einer einfachen JSON-Datenstruktur auf die Festplatte geschrieben.

Das Parsen des Befehlszeilenarguments ist Teil des `cli.py` Moduls. Die `Hiding Technique Wrapper` befinden sich im Root-Modul. Sie übernehmen die Umwandlung von Eingabedaten in Streams, sowie das Casten, Lesen und Schreiben von *Hiding Technique*-spezifischen Metadaten und den Aufruf der geeigneten Methoden der Implementierung entsprechend der gewählten Verstecktechnik.

Um das Dateisystem eines bestimmten Images zu erkennen, nutzt der `Hiding Technique Wrapper` den `filesystem_detector`, welcher wiederum Dateisystem-Erkennungsmethoden verwendet, die im jeweiligen Dateisystemmodul implementiert sind. Die Dateisystem-spezifischen Implementierungen realisieren die eigentliche Funktionalität. Sie bieten mindestens jeweils eine `write`, `read` und `clear` Methode zum Verstecken, Lesen und Löschen von Daten im Dateisystem. Sie arbeiten außerdem nur mit Streams, um eine hohe Wiederverwendbarkeit des Codes zu gewährleisten. `Hiding Technique` Implementierungen verwenden

entweder `pytsk3` oder eigene Dateisystem-Parser des jeweiligen Dateisystempakets um Informationen zum gegebenen Dateisystem zu ermitteln.

Das **CLI** bildet die Benutzeroberfläche für das vorgestellte Toolkit. Jede *Hiding Technique* ist über einen bestimmten Unterbefehl zugänglich, der selbst weitere Optionen definiert. Das CLI kann zu versteckende Daten entweder von `stdin` oder aus einer Datei lesen. Beim Auslesen versteckter Daten können diese entweder in `stdout` oder in eine bestimmte Datei geschrieben werden. Das CLI-Modul übernimmt die Analyse des Kommandozeilenarguments und ruft abhängig vom Unterbefehl den entsprechenden `Hiding Technique Wrapper` auf. Sofern Daten aus einer Datei gelesen werden, ist das CLI-Modul für die Pufferung und Weiterverarbeitung innerhalb der einzelnen `Hiding Technique´s` verantwortlich.

Jede Art von Verstecktechnik – zunächst völlig unabhängig vom verwendeten Dateisystem – hat einen eigenen *Wrapper* (z.B. das Verstecken von Daten im File Slack). Dieser **Hiding Technique Wrapper** wird vom CLI aufgerufen. Je nach Dateisystem ruft der Wrapper die spezifische `Hiding Technique` auf. Methoden zum Lesen und Löschen versteckter Daten erfordern einige Metadaten, die während einer Schreib- bzw. Versteckoperation gesammelt werden. Der `Hiding Technique Wrapper` ist also auch dafür verantwortlich spezifische Metadaten für die Lese- und Schreibmethoden bereitzustellen. Wenn der Nutzer versteckte Daten in eine Datei schreiben möchte, anstatt sie nach stdout auszugeben, ist der `Hiding Technique Wrapper` zudem für das Öffnen und das Schreiben dieser Datei verantwortlich.

Wenn eine Verstecktechnik auf bestimmte Metadaten angewiesen ist (beispielsweise um versteckte Daten fehlerfrei wiederherzustellen), muss sie diese Metadaten selbst bereitstellen. Dies erfolgt mit einer spezifischen Metadaten-Klasse, welche im Versteckprozess dazu verwendet wird, alle relevanten Metadaten zu speichern. Die Methode `write` muss diese Metadaten zurückgeben, damit der `Hiding Technique Wrapper` sie serialisieren und an die `read` und `clear` Methode weitergeben kann. Wenn eine der `write` Methoden fehlschlägt, müssen bereits geschriebene Daten unbedingt vor dem Programmende gelöscht werden. `Hiding Technique´s` können außerdem beliebige, zusätzlich benötigte Methoden implementieren.

Der **Filesystem Detector** ist eine einfache Wrapper-Funktion zur Vereinheitlichung der Aufrufe zu den Dateisystem-spezifischen Erkennungsfunktionen, die in dem entsprechenden Dateisystem-Paket implementiert sind.

Um die versteckten Daten wiederherstellen zu können, benötigen die meisten `Hiding Technique´s` zusätzliche Informationen. Diese Informationen werden in einer JSON **Metadaten-Datei** gespeichert. Die `fishy.metadata` Klasse liefert Funktionen zum Lesen und Schreiben von Metadaten-Dateien. Diese werden zudem automatisch ver- / entschlüsselt, sofern ein Passwort übergeben wird. Der Zweck dieser Klasse ist das Sicherstellen einer ähnlichen Datenstruktur aller Metadaten-Dateien. Somit kann das Programm beispielsweise früher erkennen, dass die falsche Verstecktechnik zum Wiederherstellen der Daten verwendet wird.

Sobald eine neue Verstecktechnik implementiert wird, muss diese Technik ihre eigene spezifische Metadaten-Klasse implementieren. Somit kann die Verstecktechnik selbst definieren, welche Informationen für den Wiederherstellungsprozess gespeichert werden müssen. Die `write` Methode gibt diese Informationen dann an die Wrapper-Klasse, welche die Daten serialisiert und speichert.

Tab. 1: Übersicht über die aktuell unterstützen Verstecktechniken

Kommando	Dateisystem			Beschreibung	K	E	S
	FAT	NTFS	EXT4				
`fileslack`	✓	✓	✓	Ausnutzung des File Slack	●	◐	○
`mftslack`		✓		Ausnutzung des MFT Entry Slack	●	●	○
`addcluster`	✓	✓		Zuordnung zus. Cluster zu einer Datei	●	◐	○
`badcluster`	✓	✓		Bad Cluster Zuordnung	●	◐	●
`reserved_gdt_blocks`			✓	Ausnutzung der Reserved GDT Blocks	◐	●	◐
`superblock_slack`			✓	Ausnutzung des Superblock Slack	◐	●	●
`osd2`			✓	Nutzung der osd2 Bytes in Inodes	○	◐	●
`obso_faddr`			✓	Nutzung des Inode Feldes obso_faddr	○	◐	●

Kapazität (K), Erkennbarkeit (E) und Stabilität (S) (○=niedrig; ◐=mittel; ●=hoch)

3.2 Verstecktechniken

Das Command-Line Interface unterteilt alle Verstecktechniken in separate Kommandos. Eine Übersicht mit allen derzeitig verfügbaren Kommandos für die unterschiedlichen Dateisysteme sowie deren Abschätzung anhand Kapazität, Erkennbarkeit und Stabilitätkann kann der Tabelle 1 entnommen werden. Jedes dieser Kommandos bietet die Schalter `read`, `write` und `clear` zum Lesen, Schreiben und Löschen von Daten in der entsprechenden Datenstruktur des Dateisystems. Neben den Verstecktechniken selbst, bietet das Framework zusätzliche Funktionen zur Informationsgewinnung:

- `fattools`: Um Informationen über ein FAT-Dateisystem zu erhalten kann das Subkommando `fattools` verwendet werden.
- `metadata`: Die Metadaten werden beim Schreiben in das Dateisystem in einer Datei gespeichert. Sie sind notwendig, um die versteckten Informationen zu lesen oder zu löschen. Der Befehl `metadata` ist in der Lage die gespeicherten Metadaten anzuzeigen.

In den folgenden Abschnitten werden wir die aktuell umgesetzten Verstecktechniken sowie deren entsprechenden Kommandos und ihre Funktionsweise im Detail erklären. Die detaillierte Anwendung der einzelnen Befehle befindet sich im öffentlichen, zuvor verlinkten Repository des Projekts. Zusätzlich wird für jede Technik eine Abschätzung vorgenommen, wobei jeweils die Kapazität, die Erkennbarkeit sowie die Stabilität der Technik bewertet wird. Die Kapazität beschreibt dabei die maximal erlangte Speicherkapazität der jeweiligen Technik. Die Erkennbarkeit gibt eine grobe Abschätzung der Qualität einer spezifischen Technik an bezüglich der Detektierbarkeit in forensischen Ermittlungen (begründet durch die Berücksichtigung verschiedener Dateisystem-Prüfverfahren – beispielsweise `chdsk` oder `fsck`). Zuletzt wird eine Einschätzung der Stabilität jeder Technik vorgenommen, d.h. wie wahrscheinlich eine Datenkorruption durch zukünftige Benutzerinteraktionen oder andere Dateisystemprozesse ist.

3.2.1 File Slack (FAT, NTFS, EXT4)

Das `fileslack` Subkommando verfügt über Funktionen zum Verstecken und Wiederherstellen von Daten im File Slack verschiedener Dateisysteme. Die kleinste adressierbare Einheit

auf Dateisystemebene wird *Cluster* (FAT, NTFS) oder *Block* (EXT4) genannt. Sie hat eine feste Größe, welche während der Erstellung des Dateisystems festgelegt werden kann und mit *sector_size * sectors_per_(cluster/block)* berechnet werden kann. Falls eine Datei auf dem Datenträger kleiner als die Größe eines Clusters/Blocks ist, entsteht ein ungenutzter Freiraum bis zum Beginn des nächsten Clusters/Blocks, welcher *File Slack* genannt wird und zum Verstecken von Daten genutzt werden kann.

Der *File Slack* selbst besteht aus zwei Teilen, dem *RAM Slack* und dem *Drive Slack*. Der *RAM Slack* beginnt am Ende einer Datei und reicht bis zum Ende des Sektors. Der *Drive Slack* umfasst den Bereich vom Ende des *RAM Slacks* bis zum Ende des Clusters. Mittlerweile füllen die meisten Dateisystemimplementierungen von FAT und NTFS den *RAM Slack* mit Nullen, was bei der Implementierung der Verstecktechnik berücksichtigt werden muss. Ist dieser Bereich nicht mit Nullen gefüllt, würde das auf versteckte Daten hinweisen.

Im Falle von EXT4 wird nicht zwischen *RAM Slack* und *Drive Slack* unterschieden und der gesamte Bereich mit Nullen gefüllt, was eine Erkennung versteckter Daten noch wahrscheinlicher macht. Unsere Implementierung für EXT4 beginnt daher direkt nach dem Dateiende mit dem Schreiben der Daten.

Kapazität: Hoch. Abhängig von der voreingestellten block_size, cluster_size und der Größe der Datei, welche zum Verstecken genutzt wird. Statistisch betrachtet ergibt sich die Kapazität: $(cluster\_size/2) \cdot stored\_files$.

Erkennbarkeit: Mittel. Eine Überprüfung mit `fsck.fat` ergab keine Hinweise auf versteckte Daten. Dasselbe gilt für `chkdsk` bei NTFS und `fsck.ext4` für EXT4. Die Erkennung für EXT4 ist einfacher, da der Slackbereich normalerweise nur Null-Bytes enthält.

Stabilität: Gering. Wenn sich die Größe der Original-Datei ändert, werden womöglich alle Daten überschrieben.

3.2.2 MFT Slack (NTFS)

Das `mftslack` Subkommando nutzt den Slack-Bereich von MFT-Einträgen des NTFS Dateisystems zum Verstecken von Daten. Die *Master File Table* (MFT) enthält alle nötigen Metadaten für jede Datei und jedes Verzeichnis einer NTFS Partition. Ein MFT-Eintrag füllt allerdings nicht zwingend den gesamten Bereich des ihm zugesicherten Platzes aus, was oft in einem ungenutzten Bereich am Ende eines Eintrages resultiert.

Häufig kann dieser ungenutzte Bereich noch Reste alter Einträge enthalten, die zuvor an derselben Stelle gespeichert waren[6]. Das macht den *MFT Entry Slack* zu einem unauffälligen Ort, um Daten zu verstecken. NTFS nutzt *Fixup* ([Car05], S. 253), ein Konzept zum Erkennen beschädigter Sektoren und Datenstrukturen. Wenn ein MFT-Eintrag auf den Datenträger geschrieben wird, werden die letzten zwei Bytes für eine Signatur genutzt. Um die MFT nicht zu beschädigen, ist es wichtig diese letzten zwei Bytes jedes Sektors beim Verstecken von Daten nicht zu überschreiben. NTFS speichert eine Kopie von zumindest den ersten vier MFT Einträgen (`$MFT`, `$MFTMirr`, `$LogFile`, `$Volume`) in einer Datei mit dem Namen `$MFTMirr` ([Car05], S. 219) für den Fall einer Datenkorruption. Um eine Erkennung versteckter Daten durch `chkdsk` zu vermeiden, ist es wichtig eine Kopie der versteckten Daten zusätzlich in die mit der `$MFT` korrespondierenden Einträge der `$MFTMirr` zu schreiben.

[6] Nicht der Fall mit ntfs-3g 2013.1.1.13AR.1 driver.

Kapazität: Hoch. Abhängig von der Größe der allozierten MFT-Einträge und der tatsächlichen Dateigröße. Bis zu *(allocated_size - actual_size) - 2* pro fixup Wert im Slack speicherbar.

Erkennbarkeit: Hoch. Mit den Standardoptionen wird ein Fehler in der `$MFTMirr` erkannt. Wird allerdings zusätzlich der Schalter `domirr` verwendet, erkennt `chkdsk` keine Manipulation.

Stabilität: Gering. Sofern sich die Attribute in den MFT-Einträgen ändern, werden auch die versteckten Daten überschrieben.

3.2.3 Additional Cluster Allocation (FAT, NTFS)

Cluster sind entweder unallokiert oder einer Datei fest zugeordnet. Durch das `addcluster` Subkommando können eigentlich ungenutzte Cluster einer Datei zugeordnet werden. Daraufhin wird das Dateisystem nicht länger versuchen diese Cluster zu verwenden bzw. Daten darin abzuspeichern. Dadurch entsteht beliebig viel Platz, um Daten zu verstecken.

Sollte die Datei, welcher zusätzliche Cluster zugeordnet wurden, allerdings zukünftig weiteren Speicherplatz benötigen, wird die Datei in die zusätzlich zugeordneten Cluster hineinwachsen. Dabei werden die versteckten Daten überschrieben. Daher sollte bei Verwendung dieser Verstecktechnik unbedingt eine statische Datei gewählt werden, die nicht größer werden kann.

Kapazität: Hoch. Zusätzliche Cluster-Allokationen nutzen ganze Cluster und somit ist die Technik lediglich durch *count_of_clusters* limitiert.

Erkennbarkeit: Mittel. Toos wie `fsck.fat` erkennen Cluster-Ketten, welche größer als die definierte Größe der Verzeichniseinträge sind. Unter NTFS erkennt `chkdsk` die zusätzlichen Cluster nicht, sofern die zugehörigen Attribute der Datei angepasst werden.

Stabilität: Gering. Sofern sich die Dateigröße ändert, werden die versteckten Daten überschrieben oder abgeschnitten.

3.2.4 Bad Cluster Allocation (FAT, NTFS)

Durch das `badcluster` Subkommando lassen sich Daten in sog. Bad Clustern verstecken. Sollte ein Block oder Cluster beschädigt sein, werden diese Bereiche vom Dateisystem als fehlerhaft markiert und eine Referenz auf diesen Bereich wird gespeichert.

Anschließend wird das Dateisystem diese Bereiche nicht länger für Schreibvorgänge nutzen. Markiert man eigentlich freie, nicht beschädigte Cluster als fehlerhaft, wird Speicherplatz gewonnen, um darin Daten zu verstecken. In NTFS-Dateisystemen werden diese beschädigten Bereiche im MFT-Attribut mit dem Namen `$BadClus` gespeichert. Die hier vermerkten Einträge werden daraufhin vom Dateisystem ignoriert. In FAT-Dateisystemen werden defekte Cluster in der *File Allocation Table (FAT)* vermerkt.

Kapazität: Hoch. Zusätzliche Cluster-Allokationen nutzen ganze Cluster und somit ist die Technik lediglich durch *count_of_clusters* limitiert.

Erkennbarkeit: Mittel. Das Flag ist veraltet und `fsck.fat` erkennt als *bad* gekennzeichnete Cluster.

Stabilität: Hoch. Als beschädigt deklarierte Cluster garantieren eine hohe Stabilität, da diese Kennzeichnung explizit vom Dateisystem berücksichtigt wird.

3.2.5 Reserved GDT Blocks (EXT4)

Die Reserved Blocks der Group Descriptor Table (GDT) in EXT4-Dateisystemen können durch das `reserved_gdt_blocks` Subkommando ausgenutzt werden. Die reservierten GDT Blöcke werden im normalen Betrieb solange nicht verwendet, bis das Dateisystem vergrößert wird und zusätzliche *Group Descriptors* angelegt werden.

Die reservierten GDT Blöcke befinden sich hinter der eigentlichen *Group Descriptor Table* und hinter jeder ihrer Backup-Kopien. Die Anzahl der reservierten Blöcke kann aus dem Superblock am Offset `0xCE` ausgelesen werden. Diese Verstecktechnik kann bis zu *count_reserved_gdt_blocks * count_of_blockgroups_with_copies * block_size* Bytes verstecken. Die Anzahl der Kopien variiert abhängig vom `sparse_super` Flag, welches die Kopien der *Reserved GDT Blocks* in allen Blockgruppen abspeichert, deren Gruppennummer einer Potenz mit der Basis 3, 5 oder 7 entspricht. In einem 512 MB großen Dateisystem mit einer Blockgröße von 4096 Bytes können mit dieser Technik etwa 64 * 2 * 4096 = 524288 Bytes an zusätzlichem Speicherplatz erwartet werden.

Allerdings ist diese Verstecktechnik bei einer forensischen Analyse relativ offensichtlich. Daher wird der primäre reservierte Bereich in Blockgruppe 0 übersprungen. Erst in die darauffolgenden *Reserved GDT Blocks* werden schließlich Daten versteckt. Dadurch bleibt dieses Datenversteck auch vor *e2fsck* verborgen. Zunächst werden die IDs der reservierten Blöcke aus den Informationen des Superblocks, wie etwa Gesamtzahl der Blöcke, Zahl der Blöcke pro Gruppe und Dateisystemarchitektur sowie der Zahl der reservierten GDT Blöcke unter Beachtung des `sparse_super` Flags, berechnet. Anschließend können die Daten in die jeweiligen IDs der verschiedenen Blockgruppen geschrieben werden.

Kapazität: Hoch. Diese Technik kann zum Verstecken von bis zu *reseved_gdt_blocks * block_groups * block_size* Bytes genutzt werden.

Erkennbarkeit: Hoch. Analysetools sowie ein Blick in den Hexeditor helfen dabei versteckte Daten innerhalb der Reserved GDT Blocks zu erkennen.

Stabilität: Mittel. Solange das Dateisystem nicht erweitert wird, sind Daten in diesem Bereich sicher abgelegt. Bei einer Erweiterung werden sie jedoch überschrieben.

3.2.6 Superblock Slack (EXT4)

Das `superblock_slack` Subkommando ermöglicht das Verstecken von Daten im Slack-Bereich des Superblocks. Abhängig von der Blockgröße entsteht ein annehmbarer *Slack Space* nach jeder Kopie des Superblocks in den einzelnen Blockgruppen.

Bei einer Blockgröße von 1024 Bytes ist diese Technik nicht anwendbar, da der Superblock ebenfalls eine Größe von 1024 Bytes hat. Die Anzahl der Kopien des Superblocks hängt genau wie die Kopien der Reserved GDT Blocks vom `sparse_super` Flag ab. Falls dieses Flag im Superblock gesetzt wurde, ist deutlich weniger Platz zum Verstecken verfügbar. Der verfügbare Speicherplatz dieser Technik bewegt sich im Bereich mehrerer Kilobytes. Jede Kopie bietet *block_size - 1024* Bytes an zusätzlichem Speicherplatz. Der primäre Superblock bildet allerdings eine Ausnahme, da im selben Block noch der Bootsektor Platz findet. Hier verbleiben nur *Blockgröße - 2048* Bytes zum Verstecken. Die Implementierung der Verstecktechnik im Framework sammelt zunächst unter Beachtung des `sparse_super` Flags alle Block-IDs der Superblock-Kopien jeder Blockgruppe. Daraufhin werden die Daten abhängig von der Blockgröße in die *Slack Spaces* geschrieben. Versteckte Daten profitieren von den Charakteristiken

des Superblocks, d.h. von einem sicheren Speicherbereich, denn normalerweise werden keine weiteren Daten in den *Superblock Slack Space* geschrieben. Die Technik ist allerdings, genau wie andere *Slack Space*-Verstecktechniken, für einen IT-Forensiker relativ leicht zu entdecken.

Kapazität: Mittel. Hiermit können bis zu *block_size - 2048 + superblock_copies * (block_size - 1024)* Bytes versteckt werden.

Erkennbarkeit: Hoch. Ähnlich den *Reserved GDT Blocks* werden Analysetools sowie das bloße Auge Daten in diesem Bereich entdecken.

Stabilität: Hoch. Daten im *Superlock Slack Space* werden nicht überschrieben.

3.2.7 OSD2 (EXT4)

Das `osd2` Subkommando nutzt die letzten zwei der zwölf ungenutzten Bytes des Inode-Feldes `osd2`, welches sich an Byte-Offset `0x74` in jeder Inode befindet.

Dieses Feld nutzt maximal 10 Bytes, abhängig davon ob der *Tag* `linux2`, `hurd2` oder `masix2` lautet. Daraus resultiert ein Platz von *count_of_inodes * 2* Bytes zum Verstecken. Trotz des relativ geringen Speicherplatzes, bietet diese Technik eine sichere Versteckmöglichkeit und könnte bei mehrfacher Verwendung bereits für kleinere Daten ausreichen. EXT4 validiert eine Vielzahl an Checksummen für alle möglichen Arten von Metadaten. Die Anwendung dieser Technik resultiert ohne Korrekturen zunächst in fehlerhaften Inode-Checksummen, welche anschließend neu berechnet werden müssen. In einem 235 MB bzw. 60.000 Inodes umfassenden Dateisystem können beispielsweise 120.000 Bytes versteckt werden. Um Daten zu verstecken, schreibt diese Methode die Informationen direkt in die zwei Bytes des `osd2`-Feldes. Die Adressen der verwendeten Inodes werden aus der *Inode-Tabelle* gelesen. Der Prozess wird solange fortgeführt, bis entweder keine Daten oder keine Inodes mehr verfügbar sind.

Kapazität: Gering. Pro Inode können lediglich 2 Bytes an Daten versteckt werden.

Erkennbarkeit: Gering bzw. hoch. Analysetools wie *e2fsck* bemerken veränderte Checksummen, daher können verborgene Daten relativ leicht entdeckt werden. Sobald allerdings eine Neuberechnung der Checksummen durchgeführt wird, stellt diese Technik eine sichere Möglichkeit dar, um Daten zu verstecken.

Stabilität: Hoch. Diese Felder werden vom Dateisystem nicht benutzt und werden daher nicht überschrieben.

3.2.8 obso_faddr (EXT4)

Das `obso_fadd` Subkommando verwendet das ungenutzte EXT4 Inode-Feld `obso_faddr` zum Verstecken von Daten. Bei diesem Feld, welches in jeder Inode an Offset `0x70` steht, handelt es sich um ein obsoletes Adressfeld mit einer Länge von 32 Bit.

Diese Technik funktioniert im Grunde genau wie die zuvor gezeigte *osd2*-Technik, mit der Ausnahme, dass hier die doppelte Menge an Daten versteckt werden kann. In einem 235 MB großen Dateisystem können somit bereits 240.000 Bytes an Daten versteckt werden. Für diese Technik gelten somit die gleichen Vor- und Nachteile wie zuvor.

Kapazität: Gering. Doppelt so viele Daten wie die *osd2*-Methode, daher sind *count_of_inodes * 4* Bytes an Speicherplatz verfügbar.

Erkennbarkeit: Gering bzw. hoch. Wie bei der *osd2*-Methode werden Analysetools veränderte Checksummen bemerken. Daher sollte eine Neuberechnung der Checksummen erfolgen.

Stabilität: Hoch. Diese Inode-Felder sind obsolet und daher ungenutzt. Sie werden nicht überschrieben.

3.3 Fazit

Die Menge beispielhaft implementierter Verstecktechniken bietet einen Einstieg in die Funktionalität und Verwendungsweise von `fishy`. Allgemein lässt sich feststellen, dass jedes Dateisystem in der Regel spezielle Datenstrukturen zum Verstecken von Daten bietet, die mehr oder weniger ausgenutzt werden können – sowohl in Hinsicht auf Kapazität als auch auf Erkennbarkeit und Stabilität. Doch jede Implementierung kann ihre eigenen Feinheiten mit sich bringen, ein Beispiel hierfür ist das Problem korrupter Checksummen in einem EXT4 Dateisystem. Werden einzelne Felder geändert, wie in den Verstecktechniken 3.2.7 oder 3.2.8 gezeigt, führt das zu Inkonsistenzen der Inode-Checksummen. Diese Verhaltensweisen müssen in den Dateisystemspezifischen Implementierungen der `Hiding Technique´s` beachtet werden.

4 Resümee und Ausblick

In dieser Arbeit wurde das in Python entwickelte Framework `fishy` vorgestellt. Die Implementierung integriert Schnittstellen zu den drei bekannten Dateisystemen NTFS, EXT4 und FAT. Das Framework und dessen modulare Struktur ermöglichen die zukünftige Entwicklung und Erweiterung durch zusätzliche Verstecktechniken. Im Gegensatz zu bereits veröffentlichten Tools fokussieren wir die einheitliche Bereitstellung verschiedener Techniken und deren langfristige Reproduzierbarkeit. Die modulare Struktur des Frameworks erlaubt außerdem zukünftig die Integration von anderen Dateisystemen (z.B. APFS, Btrfs, ReFS oder XFS).

Die aktuellen Entwicklungen des Frameworks bieten bereits ausreichend Funktionalitäten, um Daten innerhalb eines unterstützten Dateisystems zu verstecken. Allerdings bietet der aktuelle Entwicklungsstand noch Potential für Ergänzungen und Verbesserungen. Dazu zählen beispielsweise die direkte Integration von Dateisystem-Prüfmechanismen. Dies würde den Evaluationsprozess (z.B. mit Standard-Tools wie *fsck* oder *chkdsk*) der einzelnen Verstecktechniken weiter verbessern. Die automatische Erkennung des vorliegenden Dateisystems sollte überprüft und verbessert werden. Die aktuelle Implementierung basiert auf der einfachen Erkennung von bekannten Strings und Signaturen innerhalb des Bootsektors. *fishy* beinhaltet derzeit noch keine Funktion für eine zusätzliche Verschlüsselung oder eine Überprüfung der Integrität der Daten. Auch wenn wir dies nicht als zentrale Aufgabe unseres Frameworks in einem Forschungskontext sehen, so ist die mögliche Identifizierung von unverschlüsselten Daten offensichtlich und oftmals unabhängig von der eigentlichen Verstecktechnik selbst. Einige Ansätze zum Verstecken der Daten sind anfällig für etwaige Manipulationen der Dateien zur Laufzeit durch entsprechende Nutzung des Dateisystems. Die Stabilität der jeweiligen Techniken könnte durch die zusätzliche Integration von Integritätsprüfungen oder einer Codierung erhöht werden. Die notwendige Speicherung und Verwaltung einer Metadatei ist im Rahmen einer etwaigen Evaluation von Dateisystem-basierten Verstecktechniken notwendig, jedoch in einem wirklichen Anwendungsszenario unerwünscht. Das mögliche Verstecken der Metadaten selbst wäre ein zusätzlicher und lohnenswerter Schritt der Verschleierung. Zudem ist das CLI aktuell auf die Speicherung von nur einer Datei pro Aufruf begrenzt und berücksichtigt zuvor bereits geschriebene Dateien auf der Festplatte nicht.

Danksagung
Wir bedanken uns bei allen Teilnehmer des Bachelor-Moduls Projekt Systementwicklung, welche bei der Umsetzung des Frameworks maßgeblich beteiligt waren: Matthias Greune, Deniz Celik, Tim Christen, Dustin Kern, Yannick Mau und Patrick Naili. Für wertvolle Hinweise und Denkanstöße bedanken wir uns außerdem bei Sebastian Gärtner und Prof. Dr. Harald Baier. Diese Arbeit wurde vom Bundesministerium für Bildung und Forschung (BMBF) sowie vom Hessischen Ministerium für Wissenschaft und Kunst (HMWK) im Rahmen von CRISP (www.crisp-da.de) gefördert.

Literatur

[ANS98] R. Anderson, R. Needham, A. Shamir: The Steganographic File System. In *International Workshop on Information Hiding*, pages 73–82. Springer, 1998.

[BHS08] H. Berghel, D. Hoelzer, M. Sthultz: Data Hiding Tactics for Windows and Unix File Systems. *Advances in Computers*, 74:1–17, 2008.

[Car05] B. Carrier: *File system forensic analysis*. Addison-Wesley Professional, 2005.

[CBB16] K. Conlan, I. Baggili, F. Breitinger: Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy. *Digital investigation*, 18:S66–S75, 2016.

[EJ05] K. Eckstein, M. Jahnke: Data Hiding in Journaling File Systems. In *DFRWS*, 2005.

[Fai12] K.D. Fairbanks: An analysis of Ext4 for digital forensics. *Digital investigation*, 9:118–130, 2012.

[FL05] J.C. Forster, V. Liu: catch me, if you can... *BlackHat Briefings. Available at* http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf, 2005.

[Gar07] S. Garfinkel: Anti-forensics: Techniques, detection and countermeasures. In *2nd International Conference on i-Warfare and Security*, volume 20087, pages 77–84, 2007.

[GB18a] T. Goebel, H. Baier: Anti-forensic capacity and detection rating of hidden data in the ext4 filesystem. In *Advances in Digital Forensics XIV*, volume 14. Springer, 2018.

[GB18b] T. Goebel, H. Baier: Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. *Digital Investigation*, 24:S111 – S120, 2018.

[Gru05] The Grugq: The art of defiling: defeating forensic analysis. In *Blackhat briefings 2005*, Las Vegas, NV, August 2005.

[Har06] R. Harris: Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *Digital Investigation*, 3:44–49, 2006.

[HBW06] E. Huebner, D. Bem, C.K. Wee: Data hiding in the ntfs file system. *digital investigation*, 3(4):211–226, 2006.

[KN07] A. Krenhuber, A. Niederschick: Forensic and anti-forensic on modern computer systems. *Johannes Kepler Universität Linz*, page 11, 2007.

[LWA08] J.R. Lyle, D.R. White, R.P. Ayers: Digital forensics at the national institute of standards and technology. *National Institute of Standards and Technology, Interagency Report (NISTIR)*, 7490, 2008.

[MK99] A.D. McDonald, M.G. Kuhn: StegFS: A Steganographic File System for Linux. In *International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[NPSB07] T. Newsham, C. Palmer, A. Stamos, J. Burns: Breaking forensics software: Weaknesses in critical evidence collection. In *Proceedings of the 2007 Black Hat Conference*. Citeseer, 2007.

[NVS+16] S. Neuner, A.G. Voyiatzis, M. Schmiedecker, S. Brunthaler, S. Katzenbeisser, E.R. Weippl: Time is on my side: Steganography in filesystem metadata. *Digital Investigation*, 18:76–86, 2016.

[PDMS05] S. Piper, M. Davis, G. Manes, S. Shenoi: Detecting Hidden Data in Ext2/Ext3 File Systems. In *IFIP International Conference on Digital Forensics*, pages 245–256. Springer, 2005.

[Rid09] C.K. Ridder: Evidentiary implications of potential security weaknesses in forensic software. *International Journal of Digital Crime and Forensics (IJDCF)*, 1(3):80–91, 2009.

[Rog05] M. Rogers: Anti-forensics. *Lockheed Martin. San Diego, California. Available at* www.researchgate.net/profile/Marcus_Rogers/publication/268290676_Anti-Forensics_Anti-Forensics/links/575969a908aec91374a3656c.pdf, 2005.

[TM06] I. Thompson, M. Monroe: Fragfs: An advanced data hiding technique. *BlackHat Federal, January. Available at* http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06-Thompson-up.pdf, 2006.

[WFM13] M. Wundram, F.C. Freiling, C. Moch: Anti-forensics: the next step in digital forensics tool testing. In *IT Security Incident Management and IT Forensics (IMF), 2013 Seventh International Conference on*, pages 83–97. IEEE, 2013.