COM SYS Communication & Distributed Systems

RWTH AACHEN UNIVERSITY

Evaluation von Heuristiken zur Erkennung von Kryptoroutinen in Software

Bachelorarbeit

Felix Matenaar

RWTH Aachen University, Germany

Chair for Communication and Distributed Systems

Advisors:

Dipl.-Inform.	Réné Hummen
Dipl.-Inform.	André Wichmann
Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr. Second	Bernhard Rumpe

Anmeldedatum:	24.06.2011
Abgabedatum:	24.10.2011

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 16. Oktober 2011

Kurzfassung

Diese Arbeit beschreibt eine Evaluation von Heuristiken zur Erkennung kryptographischer Routinen in Programmen und die Konzeptionierung eines dafür notwendigen Analyseframeworks. Mit einer eigens entwickelten Software, welche in die Virtualisierungslösung Qemu integriert worden ist, werden mittels dynamischer Analyse bestehende und selbst entwickelte Heuristiken mit verschiedenen Klassen von Programmen auf ihre praktische Anwendbarkeit getestet. Diese Heuristiken messen Eigenschaften des Programmverhaltens, anhand derer sich Mengen kryptographischer Algorithmen identifizieren lassen. Das Vorgehen betrachtet im Unterschied zu vorangegangenen Arbeiten, die dieses Themenfeld ebenfalls behandelt haben, auch die Möglichkeit der Kombination von Heuristikenausgaben zum Erreichen höherer Erkennungsraten. Anhand dieser Aussagen lassen sich Sicherheitsüberprüfungen von Software, in Fällen bei denen kein Quellcode vorliegt, effizienter durchführen. Dies ist im Speziellen bei der Schadsoftware-Analyse, sowie im Allgmeinen bei Verfahren der Binäranalyse, wichtig.

Abstract

This bachelor thesis describes an evaluation of heuristics for the detection of cryptography in software. In addition the design and implementation of the therefore required analysis framework is discussed. Using this framework which is developed on top of the virtualization software Qemu, known and self-developed heuristics are tested with different classes of programms to gain knowledge about their effectiveness in practical analysis environments. The methods use dynamic program analysis to measure behavior properties of the underlying code to identify sets of cryptographic algorithms. The main difference to related work is that this thesis also evaluates the possibility of combining several heuristics to gain improved detection methods. Using these automatically generated results, one is able to lower the cost of software security analysis in cases where no source code is accessible. This applies specifically to malware analysis and the so called binary analysis in general.

Danksagung

Hiermit bedanke ich mich vor allem bei Felix Leder, der mich initial an das Thema der Programmanalyse und das Problem der Erkennung von Kryptographie in Software herangeführt hat. Ohne ihn wäre diese Arbeit nicht möglich gewesen. Weiterhin bedanke ich mich bei meinen Betreuern Réné Hummen von der RWTH Aachen und André Wichmann von der Rheinischen Friedrich-Wilhelms-Universität Bonn. Alle erwähnten Personen haben mir tatkräftig bei der Verbesserung dieser Arbeit geholfen.

Inhaltsverzeichnis

1

Einleitung

In diesem Kapitel wird zu allererst die Motivationsgrundlage der Arbeit anhand von zwei Beispielen dargelegt. Danach folgt eine Erläuterung des in dieser Ausarbeitung bearbeiteten Lösungsansatzes. Zuletzt findet eine Auflistung der Inhalte einzelner Kapitel statt.

1.1 Motivation

Im Zuge der gesellschaftlichen Entwicklung hin zur immer stärkeren Abhängigkeit von informationstechnischen Systemen bildet die Sicherung von Kommunikationswegen eine elementare Grundlage. Angewandte Verfahren sind meist standardisiert und getestet und gelten für einen bestimmten Zeitraum als sicher. Eine Umgehung der Methoden aus theoretischer Sicht ist daher nur sehr wenigen Organisationen möglich, die eher selten in Bedrohungsmodellen berücksichtigt werden. Dennoch werden in der Praxis auch ohne den Einsatz großer finanzieller Mittel kryptographisch abgesicherte Kommunikationswege erfolgreich angegriffen. Der Grund ist in fast allen Fällen entweder die Benutzung schwacher Algorithmen oder eine falsche Komposition von als sicher geltender Verfahren. Diese Problematik zeigt die Notwendigkeit der Überprüfung von Software, die zur Sicherung von Kommunikationswegen eingesetzt werden soll. Sofern der Quellcode der zu auditierenden Software vorliegt, können bekannte Methoden wie beispielsweise die statische Quellcodeanalyse angewendet werden. In Fällen, in denen diese Informationen jedoch nicht vorliegen, bedarf es Methoden, die die gewünschten - meist in Hochsprachen abgebildeten - Konzepte in der Software automatisiert erkennen. Verweigert ein Hersteller beispielsweise eine externe Überprüfung der Sicherheit, bleibt einem Abnehmer als einzige Möglichkeit, solche Methoden zu nutzen bei denen der Quellcode keine zwingende Voraussetzung darstellt.

Ähnliche Probleme existieren im Bereich der Schadsoftware-Analyse. Seit der Kommerzialisierung von Schadsoftware wird diese genutzt, um infizierte Computer in

einem eigenen Botnetz zu organisieren und deren erzeugte Daten und Ressourcen zu monetisieren. Soll nun gegen ein bereits bestehendes Netzwerk vorgegangen werden, ist die Übernahme der beteiligten Computer eine technische Möglichkeit. Nicht selten gelingt dies durch die Ausnutzung von Implementationsfehlern in der genutzten Kryptographie, welche in Botnetzen eine zentrale Rolle zur sicheren Übertragung von Kommandos des Besitzers spielt. Um solche Implementationsfehler zu finden, muss der Code der Schadsoftware analysiert werden. Der Quellcode ist dabei praktisch nie vorhanden.

Die genannten Beispiele sind nur zwei von vielen, in denen es eine Notwendigkeit zur Analyse von Software mit dem Fokus auf kryptographische Routinen gibt. Heutzutage erfolgt solch eine Analyse fast vollständig manuell als sogenanntes Reverse Engineering und ist daher in den wenigsten Fällen kosteneffizient. Eine Automatisierung oder Teilautomatisierung würde daher Analysen in vielen Fällen ermöglichen, in denen die Kosten aktuell noch zu hoch sind.

1.2 Hinleitung zum Thema

Es existieren mehrere Forschungsarbeiten zu Themenfeldern, die die Erkennung von kryptographischen Routinen zumindest als Randproblem betrachtet haben[5, 26, 42]. Bei der Erarbeitung von Methoden zum automatisierten Reverse Engineering stehen bisher jedoch fast ausschließlich Kommunikationsprotokolle von Software im Fokus, sodass als Ziele meist die Extrahierung entschlüsselter Netzwerkdaten festgelegt wurden. Im Kontext der beschriebenen Motivation reicht dies allerdings nicht aus, da Kryptographie einen sehr komplexen Bereich der Informationssicherheit darstellt, welcher weit über die reine Thematik der Verschlüsselung hinausgeht. Zur Beantwortung der Frage, ob ein gegebenes Programm kryptographische Algorithmen in einer sicheren Art und Weise verwendet, gelten folglich andere Anforderungen als jene, die in vorangegangenen Ansätzen definiert wurden. Obwohl diese Arbeiten bereits einige Grundlagen, die in diesem Themenfeld verwendet werden können, geschaffen haben, müssen diese Methoden im anderen Kontext neu überdacht und gegebenenfalls angepasst oder erweitert werden. Wie beschrieben, muss die tatsächliche Analyse relevanter Routinen manuell erfolgen; die Ausarbeitung wird sich also nicht auf die reine automatische Analyse beschränken dürfen. Deshalb sollte überlegt werden, in wie weit die Ergebnisse automatisiert aufbereitet werden können, um die darauf folgende Beurteilung durch einen Menschen zu unterstützen. Daraus ergibt sich der Vorteil, dass klare Entscheidungen der Erkennungsalgorithmen zwar wünschenswert, jedoch nicht unbedingt notwendig sind. Bei der Betrachtung von Erkennungsmethoden spielt dies eine große Rolle, da das zugrunde liegende Problem laut Satz von Rice[34] nicht entscheidbar ist. Aus diesem Grund ist mit einer Erkennung im Rahmen der Arbeit nur eine heuristische, also approximative Erkennung möglich. Korrekte Ausgaben von Erkennungsmethoden werden das Finden kryptographischer Funktionen in Software dennoch wesentlich einfacher gestalten. Anhand der erzeugten Informationen können dann Schwachstellen in Implementationen oder Kompositionen kryptographischer Algorithmen gefunden werden.

1.3 Struktur der Arbeit

Der Hauptteil dieser Arbeit besteht aus mehreren Kapiteln. Da ein Verständnis des Vorgehens bei der Evaluation teilweise Erfahrung im Bereich der Programmanalyse erfordert, erklärt das Kapitel 2 alle notwendigen Begrifflichkeiten und Konzepte, die zum Verständnis der Arbeit, deren Methodik und den in der Evaluation getroffenen Aussagen notwendig sind. Das Kapitel 3 gibt einen Überblick bezüglich vorangegangener Forschungsarbeiten, die thematisch ähnliche Ziele verfolgt haben. Zusätzlich werden Arbeiten erwähnt, aus denen einige der Heuristiken konzeptionell herausgenommen wurden, um sie in der Evaluation zu testen und zu optimieren. Im Kapitel 4 wird das in der Ausarbeitung behandelte Problem definiert und verschiedene Lösungsansätze sowie zu betrachtende Teilprobleme diskutiert. In den darauf folgenden Kapiteln 5 und 6 wird der Aufbau des Evaluationssystems genauer erläutert. Dieses basiert auf der Virtualisierungssoftware Qemu und wurde für die Nutzung im Bereich der dynamischen Programmanalyse angepasst[3]. Das Evaluationssystem wird existierender Analysesoftware gegenüber gestellt, um Vor- und Nachteile herauszuarbeiten. Das darauf folgende Kapitel 7 stellt den Kern der Arbeit dar. Dort werden die Testergebnisse der Heuristiken beschrieben, aufgearbeitet und anschließend bewertet. Darauf basierend werden einige Schlussfolgerungen gezogen, die bei zukünftigen Forschungsprojekten im Bereich Programmanalyse zum Finden von Kryptographie in Software gegebenenfalls berücksichtigt werden sollten. In einer kurzen Zusammenfassung findet eine Interpretation der Testergebnisse mit Einbeziehung der Effektivität der Evaluationssoftware statt.

2

Grundlagen

Die Grundlagen dieser Ausarbeitung erstrecken sich über vier zentrale Themenfelder:

- Kryptographie
- Virtualisierung
- Programmanalyse
- Anforderungen zur dynamischen Analyse

Im Folgenden werden einige Grundlagen der verschiedenen Klassen kryptographischer Algorithmen und deren Anwendungsbereiche vorgestellt. Im Anschluss sollen die grundlegenden Konzepte der Virtualisierung vermittelt werden, da das in dieser Ausarbeitung anteilig vorgestellte Evaluationssystem maßgeblich auf die durch Virtualisierung entstehenden Möglichkeiten zur Programmanalyse aufbaut. Danach werden die Grundlagen der Begrifflichkeiten im Zusammenhang mit Programmanalysetechniken erläutert, da diese hier genutzt werden, um Eingaben für die zu evaluierenden Erkennungsalgorithmen zu erfassen. Zuletzt sollen die wichtigsten Anforderungen erläutert werden, die eine Software zur dynamischen Analyse erfüllen sollte, da die Wahl der Softwarebasis für das Evaluationssystem und die getroffenen Designentscheidungen maßgeblich auf diesen Anforderungen basieren.

2.1 Kryptographie

Die Kryptographie befasst sich mit der Problematik, informationstechnische Systeme lese- und manipulationsresistent gegenüber unauthorisierten Dritten zu konzipieren. Im Folgenden werden die benötigten Grundlagen der Kryptographie beschrieben.

2.1.1 Komposition kryptographischer Verfahren

Zur skalierbaren und effizienten Implementation sicherer Kommunikationswege werden in der Praxis verschiedene Verfahren in Kombination eingesetzt. Die Zielsetzungen der Verfahren können in drei grundlegende Eigenschaftenkategorien unterteilt werden[37]:

- Vertraulichkeit
- Authentizität
- Integrität

Die Vertraulichkeitseigenschaft wird durch die klassische Verschlüsselung erreicht. Hierbei wird der Klartext vor der Übertragung auf einem unsicheren Medium in eine für Dritte nicht sinnvoll interpretierbare Datenstruktur umgewandelt und beim Empfänger mittels eines geheimen Schlüssels wieder lesbar gemacht. Authentizität beschreibt die Eigenschaft, dass der Kommunikationspartner sicher identifiziert werden kann. Kryptographisch wird dies oftmals mittels Zertifikaten erreicht, die von einer höheren Instanz ausgestellt werden[12]. Die Kommunikationsteilnehmer sind somit in der Lage, die Herkunft der Daten eindeutig zu bestimmen. Die dritte und letzte im Kontext der Ausarbeitung wichtige Eigenschaft ist die Integrität der Daten. Selbst wenn Daten für Dritte nicht interpretierbar und einem Kommunikationspartner eindeutig zugeordnet werden können, besteht die Gefahr einer unauthorisierten Modifikation während der Übertragung. Die Integritätseigenschaft ist genau dann erfüllt, wenn solche Modifikationen vom Empfänger erkannt werden können. Je nach Anwendungsfall sind jedoch nicht alle Eigenschaften erforderlich. Beispielsweise bei der Namensauflösung im Internet ist es sinnvoll, die Authentizität und Integrität der aufgelösten Adressen zu gewährleisten. Da die Daten jedoch meist ohnehin der Allgemeinheit zur Verfügung gestellt werden sollen, ist die Vertraulichkeit keine unbedingt erforderliche Eigenschaft.

Zusammengefasst kann gesagt werden, dass die Implementation eines Kommunikationswegs, die mindestens eine, im jeweiligen Anwendungsfall erforderliche Eigenschaft nicht gewährleistet, als unsicher betrachtet werden muss. Die Ausarbeitung soll dabei helfen, einen solchen Nachweis zu erbringen. Im Folgenden wird beschrieben, durch welche Klassen kryptographischer Algorithmen die erklärten Anforderungen an sichere Kommunikationswege erfüllt werden.

2.1.2 Symmetrische Kryptographie

Symmetrische Kryptoalgorithmen werden zur Sicherstellung der Vertraulichkeit genutzt. Definition 1 zeigt die gemeinsamen Eigenschaften dieser Algorithmenklasse.

Definition 1:
Sei P die Menge der Klartexte, C die Menge verschlüsselter Klartexte und K die Menge der Schlüssel. Ein symmetrisches Kryptosystem besteht aus zwei Abbildungen[37] $E : P \times K \to C$ und $E' : C \times K \to P$
wobei $P = E'(E(P, K), K)$.

Als Beispiel für einen einfachen symmetrischen Kryptoalgorithmus soll hier die oftmals zitierte Cäsar Chiffre herangezogen werden. Sei P das Klartextalphabet, A ein Restklassenring der Größe $|P|$ und $K \in \mathbb{N}, K \leq |A|$ ein beliebiger Schlüssel. Zusätzlich wird eine bijektive Abbildung F definiert, die jedes Zeichen aus P auf ein Element in einem Restklassenring A abbildet. Bei Cäsar ist die Verschlüsselungsfunktion für ein Klartextzeichen $p \in P$ nun definiert als $F_{encrypt} = F^{-1}((F(p) + K) mod |A|)$. Entschlüsselt wird ein Zeichen $c \in P$ anschließend wieder mit $F_{decrypt} = F^{-1}((F(c) - K) mod |A|)$. Die gemeinsame Eigenschaft aller symmetrischen Verschlüsselungsverfahren ist die Notwendigkeit, dass einem Austausch verschlüsselter Daten für dessen Entschlüsselung ein Austausch mindestens eines geheimen Schlüssels vorangegangen sein muss. Bei einem sicheren Algorithmus ist der verschlüsselte Klartext vor Zugriffen Dritter geschützt, solange die Geheimhaltung des Schlüssel sichergestellt ist und der Schlüssel nicht einfach zu erraten ist. Symmetrische Verschlüsselungsverfahren besitzen den Vorteil, dass sie bei verhältnismäßig geringer Schlüssellänge ein hohes Maß an Sicherheit bieten können und eine verschlüsselte Kommunikation effizient, im Hinblick auf die erforderliche Rechenleistung, implementiert werden kann. Aus diesem Grund kommt diese Art von Algorithmen zur Herstellung der Vertraulichkeitseigenschaft von Kommunikationswegen fast immer zum Einsatz. Ein Beispiel für einen aktuell als sicher geltenden symmetrischen Kryptostandard ist der Advanced Encryption Standard[18].

2.1.3 Asymmetrische Kryptographie

Die Authentizitätseigenschaft lässt sich alleine mit symmetrischen Kryptoalgorithmen in der Praxis nicht sicherstellen. Da der geheime Schlüssel zwangsläufig dem Kommunikationspartner mitgeteilt werden muss, wäre eine Sicherstellung der Authentizität ausschließlich mit einer komplexen Schlüsselverwaltung für jeden Sender bzw. Empfänger machbar. Darüber hinaus ist eine nachträgliche und sichere Unterscheidung von Kommunikationsteilnehmern dann nicht möglich. Definition 2 beschreibt daher die Eigenschaften Asymmetrischer Kryptosysteme, die die Schlüsselverwaltung stark vereinfachen und die Prüfung der Datenherkunft ermöglichen. Somit kann die Authentizitätseigenschaft erfüllt werden.

Definition 2:
Sei P die Menge der Klartexte, C die Menge verschlüsselter Klartexte, K und K' zwei mathematisch in Beziehung stehende Schlüssel. Ein asymmetrisches Kryptosystem besteht aus zwei Abbildungen $E : P \times K \rightarrow C$ und $E' : C \times K' \rightarrow P$ wobei $P = E'(E(P, K), K')$.

Bei einem Asymmetrischen Verschlüsselungssystem werden jeweils für eine Kommunikationsstrecke mindestens zwei Schlüssel benötigt, ein privater Schlüssel K, der geheim zu halten ist sowie ein öffentlicher Schlüssel K'. K' wird dem Kommunikationspartner im Vorfeld über einen beliebigen Kommunikationskanal mitgeteilt, darf aber von einem Angreifer nicht ausgetauscht werden. Soll ein Klartext verschlüsselt werden, geschieht die Verschlüsselung mit dem öffentlichen Schlüssel des Kommunikationspartners. Dieser kann ausschließlich mit seinem zugehörigen privaten Schlüssel die Nachricht entschlüsseln. Asymmetrische Kryptosysteme lassen sich auf Basis verschiedener mathematischer Probleme konstruieren. Der Vorteil gegenüber symmetrischen Verfahren liegt darin, dass der sichere Schlüsselaustausch in der Praxis stark vereinfacht wird. Nachteile ergeben sich durch den Gebrauch längerer Schlüssel und aufwendiger Ver- und Entschlüsselungsroutinen. Aufgrund der Möglichkeit, das Problem des Schlüsselaustauschs zu vereinfachen, wird asymmetrische Kryptographie meist zum Aufbau einer sicheren Verbindung genutzt. Ein allgemein bekannter asymmetrischer Kryptoalgorithmus ist RSA[35].

2.1.4 Kryptographische Hashfunktionen

Die letzte der beschriebenen Eigenschaften zur Herstellung sicherer Kommunikationswege ist die Integrität. Bei herkömmlichen Systemen zur Übertragung von Daten wird diese oftmals in Form einer Entdeckung von Modifikationen genutzt, welche umgebungsbedingt sind. Algorithmen zur Berechnung von Checksummen wie der Cyclic Redundancy Check[31] eignen sich nicht zur Anwendung bei absichtlichen Veränderungen. Dennoch sollte der verwendete Algorithmus in der Lage sein, wie CRC, die Daten mit möglichst wenigen zusätzlichen Metadaten zu schützen. Kryptographische Hashfunktionen nach Definition 3 erfüllen diese Eigenschaft.

Definition 3:
Sei B die Menge der Datenblöcke beliebiger Länge, D die Menge der Datenblöcke einer konstanten aber beliebigen Länge. Eine surjektive Abbildung $F : B \rightarrow D$ ist eine Hashfunktion. Ein Element $h \in D$ nennt man Hash. An kryptographische Hashfunktionen F werden zusätzlich folgende Forderungen gestellt:

- Gegeben h, ist es sehr komplex ein b zu berechnen mit $F(b) = h$
- Gegeben b, ist es sehr komplex ein b' zu berechnen mit $F(b') = F(b)$

Einen Hashwert aus gegebenen Daten zu berechnen ist effizient. Eine Nachricht zu verändern, ohne dabei seinen kryptographischen Hashwert zu ändern, muss in der

Praxis aufwändig sein. Ansonsten ist der Algorithmus nicht in der Lage, die Integritätsanforderungen zu erfüllen. Für eine sichere Kommunikation werden kryptographische Hashes zusätzlich verschlüsselt. Damit wird sichergestellt, dass eine dritte Person nicht einfach den Hash - und folglich auch die Nachricht - austauschen kann.

2.1.5 Konfusion und Diffusion

Die Erstellung eines sicheren kryptographischen Algorithmus ist ein komplexes Problem, bei dem heutzutage verschiedene Angriffstechniken berücksichtigt werden müssen. Aus diesem Grund weisen bestimmte Klassen von Algorithmen Eigenschaften auf, die im Kontext der Ausarbeitung genutzt werden können, um diese zu erkennen. Konfusion und Diffusion sind zwei grundlegende Verfahren zum Verschleiern von Redundanzen[38] in Chiffretexten. Klartexte weisen diese Eigenschaft, die eine Schwäche bei der Robustheit kryptographisch abgesicherter Kanäle darstellt, sehr häufig auf. Konfusion wird in vielen Algorithmen durch Substitution erreicht und sorgt dafür, dass Zusammenhänge zwischen Klartext und Chiffretext schwieriger zu erkennen sind. Diffusion hingegen kann durch Permutation des Klartextes erfolgen und hat zum Ziel, eventuelle Regelmäßigkeiten im Klartext über mehrere Bits des Chiffretextes zu verteilen. Dadurch werden Muster im Klartext für einen Angreifer, der nur den Chiffretext kennt, schwieriger zu erkennen sein, da diese sich über die gesamten Daten verteilen. Das Ergebnis einer Implementation dieser Maßnahmen in einem kryptographischen Algorithmus sollte idealerweise sein, dass sich bei einer Änderung von einem Bit im Klartext eine Änderung von etwa 50 Prozent der Bits im Chiffretext ergibt.

2.2 Virtualisierung

Diese Arbeit beschreibt die Evaluation von Heuristiken mithilfe einer angepassten Virtualisierungssoftware. Um zu verstehen, wie die Messung der für die Erkennungsmethoden notwendigen Eingabedaten erfolgt, müssen die verschiedenen Arten von Virtualisierung im Hinblick auf deren Umfang und Möglichkeiten erläutert werden. Zur dynamischen Programmanalyse eignen sich Virtualisierungslösungen besonders gut, da die Umgebungen, sofern sie per Software implementiert werden, durch eine Modifikation dieser Software verändert und instrumentiert werden können.

Virtualisierung und Emulation Die Grenzen zwischen Virtualisierung und Emulation sind fließend. Im klassischen Sinn bezeichnet Virtualisierung die Aufteilung von Ressourcen auf mehrere Systeme, die nichts voneinander wissen. Die Emulation hingegen beschreibt eine Imitation von Komponenten, die in einer Umgebung nicht real existieren. Virtualisierungslösungen nutzen teilweise emulierte Ressourcen, um eine bessere Aufteilung zu erreichen.

Userspace-Virtualisierung Die am wenigsten umgebungssimulierende Art der Virtualisierung ist die sogenannte Userspace-Virtualisierung. Damit ist gemeint, dass ausschließlich die ausführenden Prozesse der Software außerhalb des Betriebssystemkontextes Schnittstellen erhalten, die in der eigentlichen

Umgebung nicht vorhanden sind und deshalb nachgebildet werden. Ein praktisches Beispiel ist das Programm Wine[16]. Es dient dazu, für Windows angepasste Programme unter Linux ausführen zu können. Hierzu bietet Wine dem Programm Schnittstellen, die vom Linux Kernel selbst nicht angeboten werden. Wie eine Art Wrapper werden daher die Aufrufe der Windows API in Linux-kompatible Funktionsaufrufe umgewandelt. Das Verhalten des Programms soll sich durch die Nutzung von Wine nicht ändern.

Hardware-Virtualisierung Neuere amd64-kompatible Prozessoren unterstützten die Virtualisierung mehrerer Prozessoren als Implementation in Hardware[15]. Ein sogenannter Hypervisor ist dabei den virtualisierten Instanzen übergeordnet und teilt die real vorhandenen Ressourcen auf diese Instanzen auf. Ein Implementierungsbeispiel ist die sogenannte Kernel Virtual Machine im Linux Kernel[14].

Software-Emulation Der Betriebsmodus, in dem Qemu im Kontext der Programmanalyse ausgeführt wird, nennt sich Hardware-Emulation[3]. Dies bedeutet, dass der virtuellen Maschine softwareseitig eine komplette Hardwareumgebung vorgetäuscht wird. Dies ist aufwändiger und langsamer, als beispielsweise die durch Hardware unterstützte Virtualisierung, erlaubt aber gleichzeitig ein stärkeres Eingreifen in die Arbeitsweise des virtualisierten Systems per Software. Ein Beispiel ist das Vornehmen von Änderungen an softwareseitig vorgetäuschter Netzwerkhardware, um versendete und empfangene Netzwerkpakete transparent modifizieren zu können.

Binary-Translation beschreibt die Emulation eines Befehlssatzes, indem jeder einzelne Befehl des Quellinstruktionssatzes in eine beliebige Anzahl von äquivalenten Befehlen des Zielinstruktionssatzes umgewandelt wird. Somit lässt sich beispielsweise der Befehlssatz eines RISC-Prozessors auf einem CISC-Prozessor nachbilden und umgekehrt. Ein Vorteil der Binary-Translation liegt darin, dass der übersetzte Code auf dem Prozessor ohne weitere Abstraktionsebenen ausgeführt werden kann. Da Programme in der Regel viele Codestellen mehrmals ausführen, kann die Gesamtlaufzeit trotz des anfänglichen Übersetzungsprozesses im Vergleich zur Emulation jeder einzelnen Instruktion verringert werden, sofern die übersetzten Codestellen in einem Cache verwaltet werden. Qemu nutzt einen Binary-Translator, sofern es ohne Virtualisierungsunterstützung der Hardware ausgeführt wird.

Befehlssatz-Emulation Im Gegensatz zur meist effizienteren Binary-Translation, kann ein Befehlssatz beim Einsatz von Virtualisierungssoftware ebenfalls emuliert werden. Dabei implementiert die virtualisierende Software jeden einzelnen Prozessorbefehl und führt den entsprechenden Emulationscode aus. Die x86-Emulationssoftware Bochs[23] nutzt im Gegensatz zu Qemu einen Befehlssatz-Emulator.

Je nach Virtualisierungsimplementation werden mehrere der erwähnten Techniken in Kombination genutzt. Die erklärten Begriffe sind zum Vergleich der für die Evaluation in Frage kommenden Virtualisierungslösungen relevant. Weiterhin basiert

die Funktionsweise der Datenmessung maßgeblich auf den Virtualisierungsmethoden von Befehlssätzen (Binary-Translation). Probleme im Hinblick auf Verhaltensänderungen zu analysierender Programe in virtuellen Umgebungen werden in den Anforderungen an eine Plattform zur dynamischen Analyse näher erläutert.

2.3 Programmanalyse

Diese Ausarbeitung behandelt die Evaluation von Erkennungsmethoden kryptographischer Funktionen in Software. Hierzu existieren Verfahren der Programmanalyse, die in dieser Ausarbeitung genutzt werden sollen. Welche grundlegenden Analyseverfahren es gibt und wie das Programm genau in einzelne abgeschlossene Teile zerlegt wird, deren Eigenschaften separat ermittelt werden um Aussagen treffen zu können, wird im Folgenden beschrieben.

2.3.1 Analyseverfahren

Im Wesentlichen werden Analysemethoden in statische und dynamische Verfahren unterteilt. Die statische Analyse beschreibt alle Analyseverfahren, bei denen das zu verarbeitende Programm nicht ausgeführt wird. Die dynamische Analyse hingegen nutzt das Ausführungsverhalten um Aussagen über das Programm treffen zu können.

Statische Analysen haben den Vorteil, dass sie nicht von der Ausführung des zu analysierenden Programms abhängig sind. Da hier prinzipiell alle Programmteile, die statisch beispielsweise als Programmdatei vorliegen, analysiert werden können, weist die statische Analyse weniger Probleme im Bereich der Pfadabdeckung auf. Die dynamische Analyse hingegen verarbeitet pro Ausführung jeweils einen Programmpfad. Dabei kann es passieren, dass relevante Programmteile nicht ausgeführt und deshalb auch nicht analysiert werden. In dieser Ausarbeitung werden die Methoden der dynamischen Analyse verwendet, da sich ein Programm leichter gegen eine statische Analyse absichern kann[13] und viele der Erkennungsmethoden Ausführungsdaten des Programms als Eingabedaten benötigen. Hierzu gehören beispielsweise Informationen über die durch das Programm zur Laufzeit verarbeiteten Daten.

2.3.2 Instruktionssequenz

Da sich die Erkennungsmethoden auf die Konzepte der dynamischen Analyse stützen, muss der ausgeführte Pfad in einzelne Teile zerlegt werden. Dies ist notwendig, da ein Programm bis zur Terminierung viele Milliarden Instruktionen ausgeführt haben kann. Da viele der verwendeten Erkennungsmethoden keine lineare Laufzeit besitzen, wäre eine Verarbeitung bei einer solchen Eingabelänge nicht effizient. Eine Instruktionssequenz bezeichnet im Kontext der Ausarbeitung ein Tupel von Instruktionen aus der Menge des IA32-Befehlssatzes. Die Reihenfolge der Elementanordnung im Tupel repräsentiert dabei die Ausführungsreihenfolge durch den Prozessor. Aus vorher genannten Gründen muss diese Sequenz in kleinere Abschnitte aufgeteilt werden. Diese einzeln zu analysierenden Programmteile werden Routinen genannt und im Folgenden beschrieben.

2.3.3 Routine

Routinen bezeichnen Teilsequenzen aus der Instruktionssequenz eines Programmdurchlaufs. Routinen werden durch eine Teilmenge der im IA32-Befehlssatz definierten Instruktionen getrennt, die lokale Speicherkontexte erstellen und abbauen. Abbildung 2.1 illustriert die Trennung von Routinen aus Speicher- und Instruktionssicht. Der erste Grund zur Aufteilung der Instruktionssequenz in mehrere Routinen liegt im Vorteil der daraus resultuerenden effizienteren Verarbeitung durch die Erkennungsmethoden. Um dies zu verdeutlichen betrachten wir einen fiktiven Algorithmus zur Erkennung von Kryptographie mit der Laufzeit $n \cdot \lg n$. Teilen wir n nun in c viele gleich große Tupel n' auf und wenden den Algorithmus jeweils auf die kleineren Tupel n' an, so ergibt sich eine neue Laufzeit von $c \cdot \frac{n}{c} \cdot \lg \frac{n}{c}$. Vereinfacht bedeutet dies $n \cdot \lg(\frac{n}{c})$, was offensichtlich eine praktische Laufzeitverkürzung darstellt. Die zweite Begründung der Aufteilung liegt in der Möglichkeit zur genaueren Lokalisierung. Ein Algorithmus, der auf die komplette Instruktionssequenz angewendet wird und entscheiden würde, ob kryptographischer Code enthalten ist, könnte keine genauere Lokalisierung vornehmen. Aus diesen Gründen wird die Instruktionssequenz in Routinen aufgeteilt, die möglichst der logischen Aufteilung in kleinere Module (Funktionen) innerhalb des Programms entsprechen sollen.

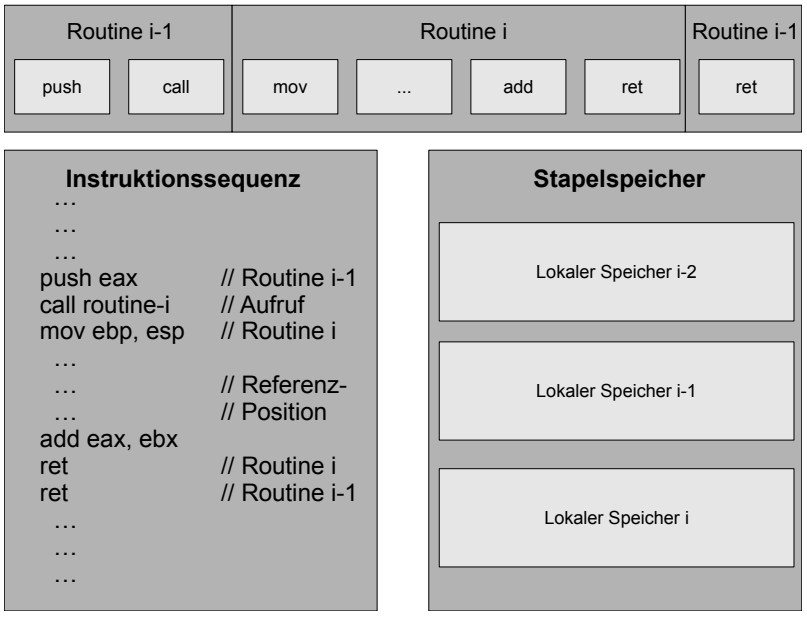

Abbildung 2.1 Abgrenzung verschiedener Routinen/Funktionen an der Referenzpostition

2.3.4 Datenquellen

Im Kontext der Ausarbeitung wird ein auf der Virtualisierungssoftware Qemu basierendes Evaluationssystem genutzt, um Eingabedaten für die zu testenden Erkennungsmethoden zu erzeugen. Dabei gibt es zwei Kategorien von Messdaten, die während einer Programmausführung ermittelt werden können. Dies sind die ausgeführten Instruktionen, die Vorgeben, wie das Programm die eigenen Daten verarbeitet.

Darüber hinaus sind die Daten messbar, die durch die Ausführung von Instruktionen bearbeitet werden. Beide Datenkategorien sind zum Aufdecken von Eigenschaften, die zur Erkennung kryptographischer Routinen in Software nachgewiesen werden müssen, von Bedeutung und können vom Evaluationssystem gemessen werden.

2.4 Anforderungen der dynamischen Analyse

Programme zur dynamischen Analyse unterliegen im Gegensatz zu herkömmlicher Software einigen speziellen Anforderungen, die besondere Konzepte zur effizienten Verarbeitung von Ereignissen erfordern. Vor allem in Bezug auf Performanz sind bei der heute zur Verfügung stehenden Rechenleistung in vielen Anwendungsfällen keine speziellen Überlegungen zu treffen. Bei der Programmanalyse müssen besonders große Datenmengen verarbeitet werden. Zusätzlich ist die Bearbeitungszeit sehr kritisch, da das zu analysierende Programm möglichst schnell ausgeführt werden soll, um Verhaltensänderungen durch zu lange Ausführungsunterbrechungen bei der Laufzeitanalyse ausschließen zu können. Da die Motivation der Ausarbeitung Analysen von Schadsoftware mit einschließt, müssen zusätzlich Überlegungen getroffen werden, eine Erkennung der Analyseumgebung durch die zu analysierende Software verhindern zu können. Im Folgenden sollen die wichtigsten Anforderungen an die genutzte Evaluationsplattform genauer dargestellt und begründet werden.

2.4.1 Minimierung der Analysezeiten

Unabhängig von der Auswertung müssen Programmdaten während der Analysezeit gespeichert und gegebenenfalls bereits aufbereitet werden. Dies kostet Rechenzeit, die je nach Anzahl zu verarbeitender Ereignisse im Verhältnis zur reinen Ausführungszeit der zu analysierenden Software sehr lang werden kann. Eine obere Schranke für zusätzliche Analysezeiten stellen Antwortzeitüberschreitungen von Netzwerkprotokollen dar, welche die zu analysierende Software gegebenenfalls implementiert und nutzt. Eine Überschreitung dieser Antwortzeiten würde die Ergebnisse des Analyseprozesses verfälschen. Im Kontext der Ausarbeitung ist die Netzwerkkommunikation zusätzlich von besonderer Bedeutung, da hier oftmals Kryptographie zum Einsatz kommt. Eine Anforderung an das Evaluationssystem ist demnach, dass zumindest bekannte und oftmals genutzte Protokolle wie TCP sowie von der Applikation selbst implementierte Protokolle ihre Antwortzeiten durch den Zusatzaufwand der Laufzeitanalyse nicht überschreiten.

Weiterhin sollte die Analysezeit nicht zu lasten der vom Betriebssystem an den zu analysierenden Prozess zugewiesenen Rechenzeit gehen, da dies die Erkennung der Analyse durch Schadsoftware vereinfacht und auch bei legitimer Software ungewollte Nebeneffekte zur Folge haben kann. Sogenannte Hooking Techniken[28], bei denen Instruktionen dem Programmcode hinzugefügt werden, um zusätzliche Auswertungsroutinen im Prozesskontext auszuführen, werden damit beispielsweise ausgeschlossen.

Da es praktisch immer möglich ist, mit einem in diesem Kontext genannten "Helfer", der einem Programm über das Netzwerk genaue Daten über tatsächliche Zeitabläufe

liefern kann, Analysezeiten durch Vergleich der Prozesszeit mit der tatsächlich vergangenen Zeit zu entdecken, kann eine Beeinflussung durch Analysen nie gänzlich ausgeschlossen werden. Bei legitimer Software wird dieses Szenario jedoch keine Rolle spielen. Es lassen sich bisher keine Beispiele finden, bei denen eine Schadsoftware solche Methoden genutzt hat.

2.4.2 Verhindern von Analyseerkennung

Mit dem Problem der Erkennung virtueller Maschinen beschäftigen sich Schadsoftware-Analysten bereits seit einiger Zeit. Während bei legitimer Software in nur wenigen Fällen Maßnahmen gegen Programmanalyse zum Einsatz kommen, kann dies bei Schadsoftware als de-facto Standard angesehen werden. Abbildung 2.2 zeigt, wie einfach eine durch VMWare virtualisierte Umgebung zu erkennen ist. Das Beispiel basiert auf der von VMWare implementierten Schnittstelle zwischen Gast- und Wirtsystem, das über Ein- und Ausgabeports der virtuellen Hardware funktioniert.

```
mov eax, 564D5868h ; Vorbereitung der Register
mov ebx, 00000000h
mov ecx, 0000000Ah
mov edx, 00005658h
in eax, dx         ; Ansprechen eines IO-Ports
cmp ebx, 564D5868h ; Auswertung der Ausgabe "VMhh"
jne @@exit         ; Verhalten bei erfolgreicher Erkennung
```

Abbildung 2.2 x86 Assembler Code zur Erkennung von VMWare Umgebungen

Ähnliche Techniken existieren für praktisch jede Virtualisierungssoftware und werden auf sehr unterschiedlichen Systemebenen angewendet. Angefangen bei Prüfungen von Einträgen der installierten Treiber in der Windows Registry reichen sie bis hin zur Analyse des Prozessorverhaltens bei - in der x86 Spezifikation - unscharf definierten Situationen. Bei hardwareunterstützter Virtualisierung unter x86-kompatiblen Prozessoren besteht zusätzlich das Problem, dass die Architektur keine wirkliche Virtualisierung zulässt, da die Relokation verschiedener Deskriptortabellen, die aufgrund des Designs bei einer virtualisierten Umgebung unabdingbar ist, von der virtuellen Maschine heraus entdeckt werden kann. Im Zeitalter von "Software as a Service" und "Cloud Computing" ist die Wahrscheinlichkeit, dass eine virtuelle Umgebung nicht zu Zwecken der Programmanalyse eingesetzt wird, jedoch stark gestiegen. Dies könnte Schadsoftwareentwickler in Zukunft vor das Problem stellen, dass Erkennungsmethoden nicht mehr anwendbar sind, da zu viele Zielsysteme dadurch ausgeschlossen würden. Zur Zeit muss durch Schadsoftware angewandte Virtualisierungserkennung jedoch Berücksichtigung finden und wird durch die von Qemu angewandte Hardware-Emulation in der Praxis ausreichend verhindert.

2.4.3 Beschreibung benötigter Datenquellen

Prinzipiell bieten Debuggingschnittstellen alle Möglichkeiten, beliebige Daten über den Ablauf eines Programms zu ermitteln. Allerdings müssten im Fall einer granularen instruktionsbasierten Analyse oftmals unnötig viele Daten abgerufen, sowie

Sprungziele berechnet werden, um Anhaltepunkte zu setzen, die für die in der Ausarbeitung genutzten detaillierten Analysen notwendig sind. Eine solche, auf Abfragen basierende Implementation, könnte den Effizienzanforderungen bei gegebener Datenmenge nicht standhalten. Optimal wäre ein auf Rückmeldungen basierendes System, welches beim Auftreten bestimmter Ereignisse die zugehörigen Analysefunktionen selbstständig aufruft. Dabei sollten auch Kontextinformationen im zu analysierenden Programmcode verwaltet werden können, um beim Auftreten eines Ereignisses zusätzlich Auswertungszeit sparen zu können. Haltepunkte sind hierfür ein gutes Beispiel. Beim Einfügen eines solchen ist die Codeadresse bereits bekannt und wäre als zusätzliche Kontextinformation sinnvoll, damit sie zur Laufzeit nicht nochmal extra abgerufen werden muss.

Im Kontext der Arbeit muss das Analyseframework Schnittstellen zur Reaktion auf folgende Ereignisse bieten:

- Kontextwechsel der Prozesse und Threads
- Identifikation des laufenden Prozesses
- Ausführung "spezieller Instruktionen"
- Speicherzugriffe
- Beginn/Ende einer Routine
- Erreichen eines Haltepunkts
- Laden von zusätzlichem Code (Bibliotheken)
- Aufruf von als relevant definierten Funktionen

Die meisten der genannten Ereigniskategorien sind selbsterklärend. Sie implizieren jedoch teilweise, dass Schnittstellen existieren, mit denen definiert werden kann, wann solche Ereignisse auftreten, beispielsweise die Definition "spezieller Instruktionen" oder die Festlegung von Haltepunkten. Zu der Menge relevanter Funktionen gehören in diesem Kontext verschiedene Routinen, mit denen die zu analysierende Software über ihren eigenen Prozesskontext hinaus Einfluss auf das System nehmen kann. Beispielsweise bietet Windows eine von Schadsoftware oft benutzte Funktion "CreateRemoteThread"[29] aus der Windows API. Sie erlaubt es einem Prozess, einen neuen Thread im Kontext eines fremden Prozesses zu erstellen, sofern dieser die Rechte eines Debuggers besitzt. In der Vergangenheit wurde die API Funktion häufig in Kombination mit weiteren Funktionen zum Kopieren von Speicher über Prozesskontexte hinaus von Schadsoftware genutzt, um für Benutzer und Antivirensoftware schwerer erkennbar zu sein. Ein Analyseframework sollte also in der Lage sein, solche Ereignisse zu registrieren und gegebenenfalls ab diesem Zeitpunkt den beeinflussten Prozess zusätzlich in seinem Verhalten zu beobachten. Es ist also von Vorteil, dass Situationen, die ein Ereignis produzieren sollen, zur Laufzeit beliebig neu festgelegt werden können.

2.4.4 Zurücksetzen von Änderungen

Da Programme außerhalb rein mathematischer Berechnungen oftmals dazu dienen, Seiteneffekte in Ihrer Umgebung zu erzeugen, muss die Beeinflussung der virtuellen Umgebung zur Laufzeit des zu analysierenden Prozesses bei der Konzeptionierung einer Analyseplattform berücksichtigt werden. Während die Vorgänge bei der Ausführung legitimer Programme in der Praxis eher selten Auswirkungen auf andere zu analysierende Programme haben, greift Schadsoftware teilweise sehr stark in die internen Abläufe eines Betriebssystems ein. Eine Analyseumgebung muss daher ohne großen Aufwand auf einen unbeeinflussten Zustand zurückgesetzt werden können, nachdem die Analyse beendet wurde. Sofern eine bestehende Virtualisierungslösung genutzt wird, kann davon ausgegangen werden, dass diese bereits Momentaufnahmen eines virtualisierten Systems erstellen und wiederherstellen kann. Bei reiner Prozessvirtualisierung kann dazu übergegangen werden, diese in einer komplett virtualisierten Umgebung durchzuführen, um ebenfalls von einer bestehenden Lösung profitieren zu können.

2.4.5 Minimierung benötiger Ausführungsdurchläufe

Es wurde bereits erwähnt, dass zu hohe Analysekosten ungewollte Beeinflussungen des zu analysierenden Programms zur Folge haben können. Trotzdem muss ein Analyseframework den Bedarf der von den Auswertungsmethoden benötigten Messdaten abdecken können. Eine Möglichkeit, falls die Verarbeitungsgeschwindigkeit nicht ausreicht, wäre dabei eine mehrmalige Ausführung des zu analysierenden Programms, wobei jeweils nur die von einer Heuristik benutzten Daten gesammelt werden. Dies würde allerdings die Vergleichbarkeit der Auswertungsergebnisse stark beeinträchtigen, da Programme, die viel Interaktion mit Prozessen außerhalb des eigenen Kontextes betreiben, mit hoher Wahrscheinlichkeit pro Ausführung jeweils andere Pfade ausführen werden.

Daraus ergibt sich die Anforderung, dass möglichst alle Messdaten innerhalb einer einzigen Ausführung gesammelt und zumindest bis zum Zeitpunkt der Auswertung durch die jeweilige Heuristik vorgehalten werden müssen. Grundlegend gibt es dabei zwei Methoden: Zum einen können die gesammelten Daten während der Ausführung des zu analysierenden Programms verarbeitet werden. Zum anderen besteht die Möglichkeit, die notwendigen Daten - aufgrund der meist hohen Menge - auf persistenten Medien zu speichern und die Auswertung der Heuristiken im Anschluss an die Ausführungszeit auszuführen. Ein gutes Konzept zur Implementation einer Analysesoftware sollte dieses Problem für jeden Verarbeitungsweg individuell betrachten, um somit die Analysezeit während der Ausführung des Programms möglichst kurz zu halten. Dies liegt daran, dass Auswertungsalgorithmen sowie die Menge an Rohdaten unterschiedlich komplex sein können.

2.4.6 Plattformunabhängige Analysen

Eine überwiegende Menge aller Anwenderprogramme wurde bisher als Userspace-Prozess in einem Windows Betriebssystem und einem IA32-kompatiblen Prozessor

ausgeführt. Mehrere Entwicklungen weisen jedoch darauf hin, dass sich dies in den nächsten Jahren zunehmend ändern wird. Durch die massenhafte Verbreitung von Smartphones sind viele Softwareentwickler auf die neuen Plattformen wie Android oder iOS umgestiegen. Damit ändert sich nicht nur das in der Softwareumgebung vorhandene Betriebssystem, sondern auch die unterstützten Prozessor-Architekturen. Um zukünftigen Einschränkungen in der Einsetzbarkeit einer Analyseplattform vorzubeugen, sollte diese zumindest so entwickelt werden, dass bei einer Portierung auf ein anderes Betriebssystem oder einer Prozessor-Architektur nur einzelne Teile der Implementierung ausgetauscht werden müssen. Hinzu kommt, dass aufgrund der komplexer werdenden Hardware-Schnittstellen von Computern auch Treiberimplementierungen zum Sicherheitsproblem werden. Da diese Art von Software in monolithischen Kernel-Architekturen nicht als Benutzerprozess ausgeführt werden, sollte eine Analyseplattform ebenfalls in der Lage sein, Programme im Kontext des Kernels zu analysieren, um mit dieser Entwicklung Schritt halten zu können.

3

Verwandte Arbeiten

Es existieren bereits verschiedene Ansätze, um kryptographische Routinen in Software automatisiert zu erkennen und gegebenenfalls zu identifizieren. Dieses Kapitel beschreibt die verschiedenen Implementationen in einer Unterteilung zwischen dynamischen und statischen Herangehensweisen. Dabei sei angemerkt, dass viele der verwandten Arbeiten das in dieser Ausarbeitung behandelte Problem gegebenenfalls nur angrenzend betrachten.

3.1 Statische Analyse

Viele der oft genutzten Verschlüsselungsverfahren benutzen eine große Menge an Konstanten. Diese werden verwendet, um verschiedene Anforderungen an einen sicheren kryptographischen Algorithmus erfüllen zu können. Statische Analyseprogramme nutzen diese Eigenschaft, um in ausführbaren Dateien nach solchen Konstanten zu suchen und somit oftmals sogar den verwendeten Algorithmus zu identifizieren. Eine zweite Idee, die in mehreren verfügbaren Programmen verwendet wird, ist die Erzeugung von Signaturen aus Instruktionssequenzen, die in den Kryptofunktionen bekannter Bibliotheken zu finden sind.

Einige Programme wurden als Plugins für bereits bestehende Analyse- und Debuggingsoftware wie OllyDBG als Freizeitprojekt entwickelt. Tabelle 3.1 zeigt einen Auszug der bekanntesten öffentlich verfügbaren Implementationen.

Ein weiterer Ansatz, der durch den Sicherheitsforscher Tobias Klein verfolgt wurde, ist die Suche nach RSA-Schlüsseln im Speicherabbild eines Prozesses[19]. Da diese nach dem PKCS#8[21] Standard in Form einer ASN.1[17] Datenstruktur im Speicher abgelegt werden, bilden die Schlüssel ein markantes Muster. Mit Hilfe eines ASN.1 Parsers und einer optimierten Suche, basierend auf Speicherbereichen mit hoher Entropie, gelang es ihm, RSA-Schlüssel zuverlässig aus Speicherabbildern von Prozessen zu extrahieren. Darauf basierend wäre es durch Verfolgen von Speicherzugriffen während einer dynamischen Analyse möglich, den verarbeitenden Code zu

Name	Plattform
Krypto Analyzer (KANAL)	PEiD
Findcrypt Plugin	IDA Pro
SnD Crypto Scanner	OllyDBG
Crypto Searcher	keine
Hash and Crypto Detector (HCD)	keine
DRACA	keine

Abbildung 3.1 Auszug öffentlich verfügbarer Programme, entnommen aus [10]

lokalisieren, der mit hoher Wahrscheinlichkeit Teil eines kryptographischen Verfahrens ist.

Der Vorteil der beschriebenen Methoden, Implementationen von Kryptographie mithilfe von Signaturen zu erkennen, liegt in der Möglichkeit, den verwendeten Algorithmus bei einem erfolgreichen Fund benennen zu können. Eine wichtige Limitierung ist jedoch, dass die Herangehensweise ausschließlich bekannte Algorithmen und im Fall der Benutzung von Instruktionssequenzen als Signaturen nur bereits Instanzen von fertig kompiliertem Code identifizieren kann. Bei der Nutzung verschiedener Compiler und deren Optimierungsschalter können erzeugte Instruktionssequenzen stark variieren. Weitere Probleme treten aufgrund der Anwendung statischer Analysen auf, da der in einem Programm enthaltene Code nicht unbedingt ausgeführt wird und deshalb möglicherweise irrelevant sein kann. Gerade wenn Software ganze Bibliotheken kryptographischer Funktionen in die Programmdatei integrieren, kann dies zu einem erheblichen Problem führen. Es soll angemerkt werden, dass die Suche nach Konstanten durchaus auch in einen dynamischen Analyseprozess eingebunden werden kann, um die Problematik der Erkennung von Code, der nicht relevant ist da er nie ausgeführt wird, vorzubeugen. Diese Möglichkeit wird im in der Ausarbeitung beschriebenen Analysesystem genutzt.

3.2 Dynamische Analyse

Bestehende Forschungsarbeiten zu der Thematik der Erkennung kryptographischer Routinen in Software basieren vorrangig auf den Methoden der dynamischen Analyse. Hierbei werden während der Ausführungszeit der zu analysierenden Software Daten gesammelt und diese genutzt, um Aussagen über das Programmverhalten treffen zu können. Grundlegend können die in kryptographischem Code angenommenen Eigenschaften in folgende Kategorien unterteilt werden:

1. hohe Anteile arithmetischer und bitbasierter Operationen
2. hohe Entropie der verarbeiteten Daten
3. mehrere Schleifendurchläufe
4. frühe Entschlüsselung eingehender Daten

Die erste Eigenschaft wird anhand der ausgeführten Instruktionen bestimmt. Eine Teilmenge des Befehlssatzes wird als zugehörig zur Menge der arithmetischen

und bitbasierten Operationen deklariert. Nach der Ausführung können dann anhand der prozentualen Anteile dieser Menge in der Instruktionssequenz Aussagen darüber getroffen werden, welche Teile der Sequenz kryptographischen Algorithmen entsprechen.

Die Entropiemessung hingegen benutzt sowohl die ausgeführten Instruktionen als Datenquelle als auch die verarbeiteten Daten des Programms. Einige Instruktionen leiten Routinen ein, sodass jeweils die Verarbeitung von Daten innerhalb dieser Routinen betrachtet wird. Verschiedene Varianten berechnen die Entropie einzelner Teilmengen verarbeiteter Daten, um Aussagen treffen zu können.

Die dritte Eigenschaft, die kryptographischem Code in verwandten Arbeiten zugeschrieben wurde, ist die Nutzung von Schleifen zur Ver- oder Entschlüsselung von Daten. Sie wurde hauptsächlich dazu genutzt, den Suchraum anderer Eigenschaften zu verringern, beispielsweise um Entropiemessungen nur auf Daten anzuwenden, die in Schleifen verarbeitet werden.

Die vierte und letzte Eigenschaft, die kryptographischem Code zur Ver- und Entschlüsselung zugeordnet wurde, ist die frühe Entschlüsselung nach dem Empfang der Daten durch das Programm bzw. die späte Verschlüsselung vor dem Versand. Diese Idee ist darauf zurückzuführen, dass viele Operationen auf Chiffretexten außerhalb homomorpher Kryptographie nicht anwendbar sind und daher vor einer Interpretation durch das Programm eine Entschlüsselung stattfindet.

Der erste veröffentlichte Ansatz von Wang et al.[42] sollte das Problem des Reverse-Engineerings verschlüsselter Protokolle lösen. Hierbei ging es nicht um die Erkennung der genutzten Algorithmen, sondern um die Extrahierung von Klartext-Protokolldaten, die über verschlüsselte Kommunikationskanäle in unbekannten Programmen empfangen werden. Es wurde die Beobachtung formuliert, dass kryptographischer Code im Unterschied zu anderen Verarbeitungsroutinen starken Gebrauch arithmetischer und bitbasierter Operationen macht. Diese Erkenntnis wurde zusammen mit der Annahme benutzt, dass eingehende Daten sehr kurz nach dem Empfang durch das Programm entschlüsselt werden. Durch eine Data Lifetime Analysis werden in der dynamischen Analyse empfangene Daten in ihrer Verarbeitung verfolgt und der Anteil an arithmetischen Instruktionen verarbeitender Routinen kumulativ gemessen. Die Autoren gehen davon aus, dass der kumulative Anteil arithmetischer und bitbasierender Operationen in einer anfänglichen Entschlüsselungsphase nach dem Empfang sehr hoch ist, bevor er durch Ausführung der Protokollverarbeitungsroutinen sinkt. Werden im Anschluss nun wieder Daten verschlüsselt um beispielsweise auf empfangene Pakete zu reagieren, steigt der kumulative Anteil wieder, und die entschlüsselten Daten liegen folglich zwischen dem Erreichen des Minimums und Maximums entschlüsselt vor. Diese Einschränkung auf einen kleinen Bereich verarbeitender Routinen in Kombination mit der Verfolgung der empfangenen Daten ermöglichte den Autoren somit, Klartextdaten des ursprünglich verschlüsselten Textes auszulesen.

Ein späterer Ansatz von Caballero et al.[5] bezieht sich ebenfalls auf die Problematik des Protokoll Reverse Engineerings. Hierbei nutzten die Autoren wie Wang et al. die Erkenntnis, dass Verschlüsselungs- und Kodierungsroutinen einen hohen prozentualen Anteil arithmetischer und bitbasierter Instruktionen enthalten. Kodierungs- und Verschlüsselungsroutinen wurden dabei mit einer Heuristik gefunden, die Funktionen markiert, die eine Mindestlänge von 20 Instruktionen und einen prozentualen

Anteil arithmetischer und bitbasierter Operationen von ≥ 55 % aufweisen. Diese Methode wurde eingeführt, da ein zu analysierendes Programm die Daten in mehreren Schritten entschlüsselte und die Annahmen über Minimum und Maximun von Wang et al. somit nicht mehr zutrafen. Bei der Evaluation wurde eine Software mit 22.379 unterschiedlichen Funktionen genutzt. Dabei wurden laut Aussagen alle 21 kryptographischen Funktionen richtig erkannt. Die Anzahl der fälschlicherweise erkannten Funktionen betrug 9, also eine Rate von 0.04 Prozent.

Im Jahr 2008 beschrieb Noe Lutz[26] in seinem Paper, wie er mit seinem Analysefamework automatisiert Entschlüsselungsroutinen findet. Das Ziel war es, wie in den Arbeiten von Caballero und Wang, verschlüsselte Daten in ihrer Verarbeitung zu verfolgen und somit entschlüsselt auszugeben. Benutzt wurde dabei eine Kombination aus Taint-Tracking, Entropiemessungen und der Erkenntnis, dass Implementationen kryptographischer Algorithmen arithmetische und bitbasierte Operationen nutzen. Taint-Tracking bezeichnet dabei eine Verfolgung der Verarbeitung von Daten durch die Definition von Quellen und Zielen. Anhand eines Markierungsalgorithmus wird festgestellt, ob ein der Quelle entsprungener Wert den Zielwert beeinflusst. Anders als bei den vorherigen beiden Ansätzen wird die Menge, der als Entschlüsselungsroutinen infrage kommenden Funktionen durch einen Schleifenerkennungsalgorithmus in Kombination mit der Erkennung von auf markierten Daten angewandte arithmetische Operationen, eingeschränkt. Die Methode basiert auf der Annahme, dass kryptographische Algorithmen oftmals in mehreren Durchläufen von Schleifen die zu entschlüsselnden Daten verarbeiten. Ausgegeben werden dabei alle Routinen, die sich in der beschriebenen Kandidatenmenge befinden und dessen Datenverarbeitungsprozess eine signifikante Entropieverringerung zur Folge hatte. Als Entropiemaß wurde dabei eine selbst definierte Formel einer sogenannten skalierten Entropie verwendet, um Ein- und Ausgabedaten verschiedener Längen sinnvoll vergleichen zu können.

Im Jahr 2009 beschrieb Leder et al. einen Ansatz, der sich ebenfalls auf die genannte Eigenschaft der späten Ver- bzw. frühen Entschlüsselung von Daten vor der Verarbeitung durch das Programm stützt[24]. Der Fokus lag hier ebenfalls auf Routinen, die Netzwerkverkehr von Schadsoftware durch die Anwendung von Kryptographie vor einer Analyse bzw. Erkennung schützen sollen. Die Annahme war dabei, dass kurz vor der Verschlüsselung Datenpuffer allokiert werden, die dann zur Speicherung des Chiffretextes genutzt werden, bevor dieser versendet wird und analog im Fall eingehender Daten. Allokierte Speicherbereiche wurden bis zum Versand in ihrer Verarbeitung verfolgt, um somit die Menge der Routinen, die als Kandidaten verschlüsselnder Funktionen in Frage kommen, zu reduzieren.

In einer Master Arbeit von Felix Gröbert[10] wurden mehrere Heuristiken vorheriger erwähnter Arbeiten mit neuen Methoden zusammengefasst. Ziel war es dabei, vor allem die Methoden in ein einheitliches Framework zu integrieren. Erstmals wurde das Hauptaugenmerk dabei auf die Erkennung kryptographischer Routinen gelegt und weniger auf die Problematik des automatisierten Protokoll Reverse Engineerings. Mit Hilfe einer auf PIN[25] basierenden Auswertungssoftware nutze Gröbert sowohl generische Erkennungsmethoden als auch Instruktions- und Konstantenmuster, die bestimmte Implementationen bzw. Algorithmen erkennen sollen. Zusätzlich wurde dabei versucht, verarbeitete Daten mit Referenzimplementierungen bekannter Algorithmen zu vergleichen und somit auf den verwendeten Algorithmus zu schließen. Die Evaluation der Arbeit enthielt keine Tests mit Schadsoftware. Thematisch ist

diese Ausarbeitung nah an der Arbeit von Gröbert angesiedelt und nimmt einige der beschriebenen Erfahrungen als Grundlage.

Zusammenfassend lässt sich sagen, dass bereits einige Arbeiten existieren, welche die Thematik der Erkennung von kryptographischen Routinen in Software zumindest als Teilproblem bearbeitet haben. Inwieweit die Ergebnisse für praktische Ansätze jedoch zu gebrauchen sind ist fraglich, da die Methoden oftmals sehr eng gefasst wurden und vorhandene Evaluationen mit Schadsoftware teilweise allein aufgrund von Beschränkungen der Implementationen nicht repräsentativ durchgeführt werden konnten. Diese Arbeit grenzt sich vor allem dadurch ab, dass der kryptographische Code an sich Ziel der Erkennung ist und weniger die verarbeiteten Daten. Gröbert[10] hatte ebenfalls dieses Ziel, konzentrierte sich dabei aber nicht auf die Evaluation vorhandener Erkennungsmethoden und benutzte eine einfache Einteilung in generische und spezifische Erkennungsmethoden, die nach Meinung des Autors dieser Ausarbeitung in differenzierterer Weise mehr Vergleichbarkeit zwischen den Heuristiken ermöglichen würde. Zusätzlich soll in dieser Ausarbeitung ebenfalls die Nutzung mehrerer Methoden in Kombination evaluiert werden, um somit ggf. bessere Erkennungsleistungen realisieren zu können.

4

Problembeschreibung

Die Kryptographie bezeichnet ursprünglich Techniken, mit denen Informationen in einer bestimmten Art und Weise vor dem Zugriff dritter geschützt werden können. Diese Arbeit behandelt das Problem der Erkennung von kryptographischen Routinen in Software. Da diese Beschreibung nicht genau genug ist, um zielführende Methoden zu erarbeiten und zu bewerten, soll dieses Kapitel das in der Ausarbeitung behandelte Problem genauer beschreiben und eingrenzen.

4.1 Problemstellung

Das Problem der Erkennung von Kryptographie in Software beschreibt eigentlich die Erkennung von ausgeführten Routinen, die einer Implementation eines abstrakten Konzepts entsprechen. Die Erkennung wird mittels Nachweisen von Eigenschaften in Kombination mit einer Abbildung dieser Eigenschaften auf eine Menge von Entitäten erreicht. Entitäten können dabei beispielsweise kryptographische Algorithmen in ihrer Gesamtheit, einzelne Klassen kryptographischer Algorithmen oder Implementationen eines einzelnen Algorithmus sein. Als erkannt gilt eine Routine genau dann, wenn die nachgewiesenen Eigenschaften auf eine Menge eindeutig abbildbar sind. Abbildung 4.1 illustriert den Prozess der Abbildung von Routinen auf Mengen.

4.2 Problemeingrenzung

Da Programme aus Modulen bestehen, die Implementationen verschiedener Algorithmen entsprechen, muss die Instruktionssequenz eines Programmdurchlaufs in kleinere Einheiten unterteilt werden. Diese Einteilung erfolgt in eine Menge von Routinen, für welche die Eigenschaften nachgewiesen werden müssen. Die folgende

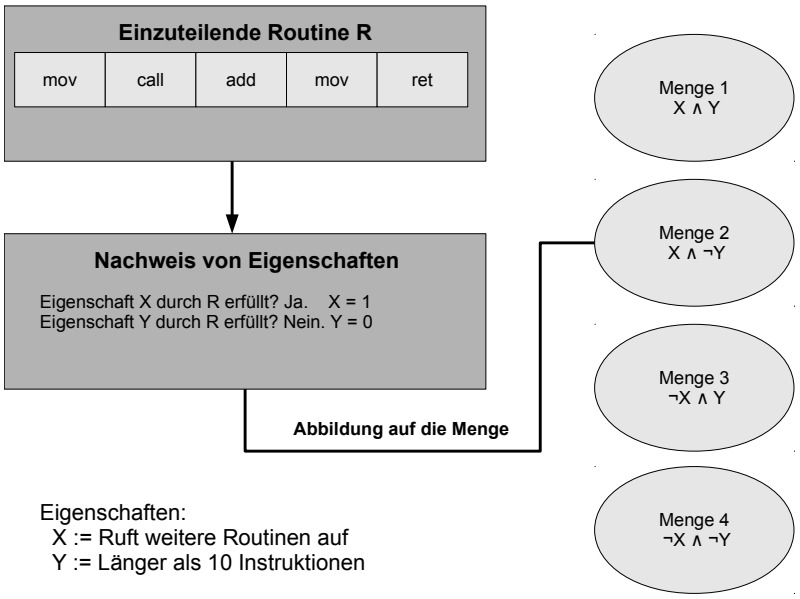

Abbildung 4.1 Abbildung einer Routine auf die Menge der repräsentierenden Eigenschaften

Definition dient der Beschränkung der Menge von Routinen, die durch die in der Ausarbeitung verwendeten Methoden erkannt werden sollen.

Definition 4:
Kryptographische Routinen, die im Kontext der Arbeit erkannt werden sollen, umfassen alle Repräsentanten von Implementierungen eines kryptographischen Algorithmus in Form von Instruktionssequenzen. Mit der Menge der *kryptographischen Algorithmen* sind moderne Blockchiffren, Hashalgorithmen sowie asymmetrische Kryptosysteme gemeint.

Diese Definition schließt Algorithmen, wie die Cäsar Chiffre oder die im zweiten Weltkrieg benutzte Enigma[20], aus. Weiterhin soll verdeutlicht werden, dass sich die Arbeit im Gegensatz zu einigen verwandten Arbeiten nicht nur auf die Erkennung von Ver- und Entschlüsselungsroutinen beschränkt. Es sollen auch Routinen erkannt werden, die, beispielsweise unabhängig von Netzwerkinteraktion, Implementationen kryptographischer Algorithmen zur Integritätssicherung von Dateien entsprechen. Dies ist sinnvoll, da die Vertraulichkeitseigenschaft allein keine hinreichende Bedingung für die Datensicherheit darstellt.

Da nun feststeht, welche Mengen von Algorithmen und abstrakten Konzepten erkannt werden sollen, fehlt noch eine Erklärung, wie die Eigenschaften nachgewiesen werden. Die Erkennungsmethoden sollen, wie in der Einleitung beschrieben, das sogenannte Reverse-Engineering einfacher und damit effizienter machen. In den jeweiligen Anwendungsfällen wird dabei davon ausgegangen, dass die zu erkennenden Strukturen wie Implementationen oder Algorithmen, also der Quellcode, im zu analysierenden Objekt, dem Programm, nicht vorhanden sind. Das Programm ist vielmehr eine Abfolge von Instruktionen, das diese Konzepte wiedergibt, wobei die Möglichkeit der Herleitung der Eigenschaften aus der ausgeführten Instruktionssequenz im Allgemeinen weder benötigt wird, noch von den Autoren der Software gewollt ist. Die Schwierigkeit, abstrakte Eigenschaften in kryptographischen Funk-

tionen zu beobachen und in den räpresentierenden Instruktionsfolgen nachzuweisen, bildet daher den Kern aller Erkennungsmethoden im Kontext dieser Ausarbeitung. Nach dem Satz von Rice[34], der besagt, dass es keinen Algorithmus geben kann, der entscheidet, ob eine Funktion durch ein Programm berechnet wird, folgt, dass auch das Problem der Erkennung kryptographischer Routinen in Software, also zumindest die allgemeine Erkennung kryptographischer Funktionen, nicht berechenbar ist. Erkennungsmethoden können also nur approximative Aussagen liefern.

4.3 Problemdifferenzierung

Wie bei der Problemstellung beschrieben werden Routinen durch die in ihnen erkannten Eigenschaften auf Mengen abgebildet. Darüber hinaus bedarf es einer differenzierten Betrachtungsweise von Erkennungsleistungen. Ein Beispiel hierfür ist die folgende Gegenüberstellung zweier Erkennungsmethoden: Angenommen eine Erkennungsmethode zur Analyse von Konstanten erkennt einen kryptographischen Algorithmus A. Dann ist die Aussage der Erkennung, dass die erkannte Routine dem Algorithmus A entspricht. Betrachten wir hingegen eine Erkennungsmethode, die Kryptographie im Allgemeinen erkennt, so wäre die Aussage, dass die erkannte Routine irgendeinem kryptographischen Algorithmus entspricht. Die erste Erkennungsmethode liefert in diesem Fall also mehr Informationen. Wird ein kryptographischer Algorithmus B ausgeführt, würde die erste Methode eine negative Aussage geben, die zweite Methode immer noch die gleiche, richtige Aussage geben können, die in diesem Fall informativer wäre. Dies erschwert den Vergleich von Erkennungsmethoden. Gröbert[10] teilte die verwendeten Methoden in die Kategorien "implementationsabhängig" und "generisch" auf. Dies reicht bei der Bandbreite an Erkennungsmethoden nicht aus.

Definition 5:
Sei M die Menge aller kryptographischer Algorithmen nach Definition 4.
Sei S die Menge aller Instruktionssequenzen.
Sei P die Menge aller Implementierungen aller Algorithmen.
Seien K eine beliebige Partitionierung von M, bei der die Intensionen der Mengen K_i jeweils eindeutig sind.
Seien $A_{(1,1)}...A_{(n,m)}$ kryptographische Algorithmen mit $A_{(i,1)} \cup ... \cup A_{(i,n)} = K_i$.
Seien $I_{(1,1,1)}...I_{(n,m,o)}$ alle kryptographischen Implementierungen
mit $I_{(i,j,k)} \in P$ und $I_{(i,j,1)} \cup ... \cup I_{(i,j,o)} \hat{=} A_{(i,j)}$.

Folgende vier Probleme lassen sich im Kontext der Ausarbeitung definieren:
Allgemeines Erkennungsproblem:
$Pg := \{x | x \in S, x \hat{=} a, a \in M\}$
Klassenspezifisches Erkennungsproblem:
$Pk_i := \{x | x \in S, x \hat{=} a, a \in K_i\}$
Algorithmenspezifisches Erkennungsproblem:
$Pa_{(i,j)} := \{x | x \in S, x \hat{=} a, a \in A_{(i,j)}\}$
Implementationsspezifisches Erkennungsproblem:
$Pi_{(i,j,k)} := \{x | x \in S, x \hat{=} a, a \in I_{(i,j,k)}\}$

Mit diesem Hintergrund ist Definition 5 zu betrachten. Diese legt vier Teilprobleme fest, nach deren Lösungen die in der Ausarbeitung zu evaluierenden Erkennungsmethoden kategorisiert werden können. Jede in der Ausarbeitung behandelte Erkennungsmethode lässt sich als Lösung eines Problems der vier definierten Erkennungsprobleme betrachten. Die Kategorisierung erlaubt eine Kombination von Erkennungsmethoden, um Erkennungsleistungen sowie -Aussagen zu verbessern. Dies könnte dadurch geschehen, dass beispielsweise eine der Klasse "Hashalgorithmen" zugewiesene Routine nur noch von Lösungen des algorithmenspezifischen Erkennungsproblems verarbeitet werden, die einen Hashalgorithmus erkennen, um diesen genauer zu identifizieren. Sofern die Klasseneinteilung gut funktioniert, würde dies eine Eingrenzung der Anzahl falscher Positive zur Folge haben, da Erkennungsmethoden, die beispielsweise Blockchiffren erkennen, nicht zur Anwendung kämen und somit keine falschen Aussagen treffen können.

Dass alle Problemkategorien in der Praxis vorkommen und daher sinnvoll sind, soll im Folgenden erläutert werden. Eine Lösung des Allgemeinen Erkennungsproblems ist in Situationen notwendig, in denen ein kryptographischer Algorithmus verwendet wird, der vorher unbekannt ist und nicht die Eigenschaften einer in anderen Klassen zusammengefassten Algorithmen teilt. Dieser Fall tritt eher selten auf und kann in der Ausarbeitung nicht repräsentativ evaluiert werden. Falls sich herausbilden sollte, dass eine Methode im Hinblick auf die restlichen Erkennungsprobleme übergreifend eine gute Erkennungsleistung besitzt, könnte eine gute Approximation der Lösung des Allgemeinen Erkennungsproblems gefunden worden sein. Eine solche Methode kann dabei helfen, die Menge zu verarbeitender Routinen durch Lösungen anderer Erkennungsprobleme stark zu reduzieren. Lösungen des Klassenspezifischen Erkennungsproblems erkennen Mengen von Algorithmen, die bestimmte Eigenschaften teilen und sich von anderen Algorithmen dadurch abgrenzen. Sofern beispielsweise alle Blockchiffren bestimmte Eigenschaften gemeinsam haben, könnten hierdurch mit hoher Wahrscheinlichkeit auch unbekannte Blockchiffren erkannt werden, sofern die Eigenschaften darauf zutreffen. Da eine genaue Identifikation des Algorithmus im Allgemeinen beim Reverse-Engineering erwünscht ist, um direkte Klarheit über die Funktionsweise zu bekommen, wurde das algorithmenspezifische Erkennungsproblem definiert. Die Limitierung ist hierbei jedoch, dass der zu erkennende Algorithmus im Vorfeld bekannt sein muss, um diesen erkennen zu können. Ist dies nicht möglich, da beispielsweise ein eigens entwickelter Algorithmus zum Einsatz kommt, der aber durch eine Lösung der beiden zuvor beschriebenen Problemkategorien erkannt wurde, ist zusätzliche Arbeit zum Erhalt eines genaueren Verständnisses erforderlich.

Es existieren einige Implementationen kryptographischer Algorithmen, die bekannte Schwachstellen aufweisen. Da Entwickler teilweise auf solche Implementationen zurückgreifen, ist eine Erkennung dieser Implementationen für die in der Motivation beschriebenen Anwendungsfälle gegebenenfalls von großem Nutzen. Fehlerhafte Implementationen könnten also mit den Lösungen implementationsspezifischer Erkennungsprobleme gefunden werden.

Die hier vorgestellte Betrachtungsweise ist eine gute Grundlage zur Evaluation der Erkennungsmethoden, da über die reine Ermittlung von Richtig- und Falschmeldungen zusätzlich die Aussagekraft sowie Anwendbarkeit der Heuristiken in Betracht gezogen werden kann. Zusätzlich werden die Methoden im Hinblick auf eine Verbesserung der Erkennungsleistung bei Nutzung der Verfahren in Kombination evaluiert.

5

Design

Das Evaluationssystem, mit dessen Hilfe in dieser Ausarbeitung Methoden zur Erkennung von Kryptographie in Software entwickelt und bewertet werden sollen, bietet die notwendigen Grundlagen für eine repräsentative Testphase. Obwohl bereits mehrere Frameworks im Bereich der dynamischen Programmanalyse existieren, war es notwendig, einen für den speziellen Anwendungszweck angepassten neuen Ansatz zu verfolgen. Dazu gehören ebenfalls die im Kontext der Ausarbeitung entwickelten und angepassten Erkennungsmethoden. Was die Erstellung der verwendeten Analysesoftware angeht, so wurde ein Großteil der Implementation im Kontext einer vorherigen Projektarbeit erreicht und fällt nicht in den Zeitraum der Bearbeitungszeit. Die Implementation der Heuristiken sowie die Konzeptionierung der Datenaggregation und Auswertung wurde während der Ausarbeitung geleistet.

5.1 Konzeptionierung des Evaluationssystems

Zur Erkennung von Kryptographie in Software bedarf es Methoden, die durch Eingaben in Form von Messdaten verwertbare Aussagen zur Lösung der definierten Probleme treffen. Um Messdaten über ein breites Spektrum an Programmen ermitteln zu können, muss das Evaluationssystem, das die Messungen durchführt, die bereits im Grundlagenkapitel genannten Anforderungen erfüllen. Das Konzept, mit dem dieses Ziel erreicht wurde, soll im Folgenden genauer erklärt werden.

5.1.1 Auswahl der dynamischen Analyse

Die hier beschriebene Auswertungssoftware nutzt die Konzepte der dynamischen Analyse. Dies bedeutet, dass das zu analysierende Programm zur Datenerhebung ausgeführt wird. Im Gegensatz zur dynamischen Analyse können Programme auch

statisch, also ohne dessen Ausführung, betrachtet werden. Es existieren jedoch verschiedene Gründe, weshalb Programme vor einer statischen Analyse geschützt werden. Bei Schadsoftware ist der Zweck oftmals, die Ermittlung von Gegenmaßnahmen zu verzögern. Ein effektiver Schutz vor statischen Analysen ist dabei sehr einfach durch das Entpacken oder Entschlüsseln des eigentlichen Codes beim Starten des Programms oder sogar noch später zur der Laufzeit implementierbar. Aus diesem Grund nutzen die in den verwandten Arbeiten entwickelten Erkennungsmethoden Messdaten, die nur bei der Ausführung des Programms erzeugt werden können. Somit muss die Evaluationsplattform mittels einer dynamischen Analyse Messdaten sammeln. Im Folgenden soll erklärt werden, warum die Nachteile der dynamischen Analyse in der Praxis ein eher geringes Problem darstellen:

Im Kontext von Schadsoftware muss berücksichtigt sein, dass diese Art von Software oftmals sehr eng formulierte Aufgaben, wie zum Beispiel den Versand von Spam-Emails oder das Abgreifen von Bankdaten, bearbeitet. Weiterhin sind, durch die fehlende Interaktion mit dem Benutzer, die ausgeführten Arbeitsschritte fast vollständig durch den Programmierer im Vorfeld festgelegt. Legitime Programme besitzen diese Eigenschaften mit einer wesentlich geringeren Wahrscheinlichkeit. Dafür ist die Grundfunktionalität jedoch meist weitaus besser bekannt, sodass bei dieser Art von Programmen in vielen Fällen eine Beeinflussung des Benutzers auf die ausgeführten Routinen praktikabel ist. Es könnten somit vermeintlich relevante Programmpfade durch die gezielte Nutzung der zu analysierenden Software angesprochen werden. Gelten diese Argumente bereits auf einer allgemeinen Ebene der dynamischen Programmanalyse, so ist bei der Erkennung kryptographischer Routinen hinzuzufügen, dass die zugrunde liegenden Konzepte zum Großteil bei der Kommunikation zwischen Endsystemen unabhängig vom Inhalt angewendet werden. In solchen Fällen wäre die Ausführung von Routinen, die eine Kommunikation implementieren, ausreichend, um ebenfalls die Ausführung der dafür zuständigen kryptographischen Routinen zu erreichen.

5.1.2 Auswahl der Virtualisierungssoftware

Die Analysesoftware zur Evaluation der in der Ausarbeitung behandelten Erkennungsmethoden besteht aus einer Reihe von Anpassungen der Virtualisierungssoftware Qemu. Dies ist ein Prozessor-Emulator für IA32-kompatible Plattformen, der mithilfe einer Übersetzung von fremden Befehlssätzen in den nativen Befehlssatz, genannt dynamic Binary-Translation, eine effiziente Lösung zur Vollvirtualisierung, ohne die Notwendigkeit zur Unterstützung von Virtualisierung durch die Hardware, darstellt. Zum einen wird dadurch die virtualisierungsseitige Anforderung an Performanz im Vergleich zur Befehlssatzemulation besser erfüllt. Zum anderen erleichtert die Befehlsübersetzung die Portierung des Evaluationssystems auf andere Prozessor-Architekturen.

Wie eine Evaluation der Universität Berkeley zeigt, verhält sich Schadsoftware bei einer erfolgreichen Erkennung einer virtualisierten Umgebung nicht selten anders als auf einem normalen Computersystem[44]. Die im Evaluationssystem genutzte Vollvirtualisierung erweist sich in diesem Kontext als weitaus robuster gegen praktisch angewandte Erkennungsmethoden von solchen Umgebungen. Somit ist die Anforde-

rung an Schutz vor Virtualisierungserkennung im Vergleich zu anderen Lösungen, wie VMWare oder KVM, durch Qemu besser erfüllt.

5.1.3 Vergleich zu anderen Plattformen

Bisher fehlt die Begründung, warum die Evaluation auf Basis eines eigenen Instrumentierungsframeworks vollzogen wurde. Abbildung 5.1 zeigt einen Vergleich bekannter und öffentlich Verfügbarer Software, welche für die Evaluation in Frage kamen. Die Implementationen können in drei verschiedene Virtualisierungskategorien unterteilt werden:

1. Prozessvirtualisierung
2. Hardwarevirtualisierung
3. Vollvirtualisierung

Die Kriterien für die folgenden Vergleiche beziehen sich auf die im Grundlagenkapitel erläuterten Anforderungen an eine Plattform zur dynamischen Programmanalyse, also der Portierbarkeitsaufwand zwischen verschiedenen Befehlssätzen, Performanz der Virtualisierungsmechanismen, Unabhängigkeit von dem genutzten Betriebssystem in der virtuellen Maschine sowie die Möglichkeit zur Verhaltensanalyse der vom virtualisierten System genutzten Hardware.

Lösungen, die Prozessvirtualisierung nutzen, wurden aufgrund der fehlenden Hardwareemulation und der Abhängigkeit des Betriebssystems ausgeschlossen. Die Arbeit von Lutz[26] mit Valgrind zeigt deutlich, dass Implementationen für unixoide Systeme zur Analyse von Schadsoftware ungeeignet sind[32]. PIN wird zwar auch von Windows unterstützt, kann von Programmen jedoch leicht erkannt werden, da es sich im gleichen Systemkontext wie das zu analysierende Programm befindet. Ein weiteres Problem bei der Nutzung von Prozessvirtualisierung im Allgemeinen ist, dass die Messung, Auswertung und Speicherung im Kontext der Analyseumgebung stattfindet. Aus diesem Grund ist die Wahrscheinlichkeit einer Beeinflussung des zu analysierenden Programms unnötig höher als bei anderen Virtualisierungsmethoden.

Lösungen wie Ether, die auf den von Intel und AMD entwickelten Erweiterungen IA32 kompatibler Prozessoren zur hardwarebasierten Unterstützung von Virtualisierung basieren, haben die soeben genannten Probleme der Abhängigkeit vom Betriebssystem nicht[8]. Dennoch wird die im Kontext der virtualisierten Umgebung bereitgestellte Hardware nur teilweise emuliert und durch einen sogenannten Hypervisor kontrolliert. Die Verarbeitung von Daten durch den Prozessor ist somit keiner softwareseitigen Instanz mehr untergeordnet. Des weiteren ist die Erkennung von Hardwarevirtualisierung leicht implementierbar[36]. Zuletzt besteht das Problem, dass nicht alle Prozessor-Architekturen Hardwarevirtualisierung implementieren und darauf basierende Lösungen somit bereits rein konzeptuell nicht portierbar sind.

Die dritte Kategorie von Virtualisierungslösungen beinhaltet Implementationen, in der die virtualisierten Systeme in einer eigenen Hardwareumgebung ausgeführt werden. Da zur performanten Messung der Programmdaten weitreichende Modifikationen notwendig sind, standen nur freie Lösungen zur Auswahl. Bekannte Implementationen sind dabei Qemu, Virtualbox und Bochs. Da Bochs kein Binary-Translation

implementiert, sondern die Befehlssätze vollständig emuliert, können somit die Performanzanforderungen nicht erfüllt werden. Virtualbox ist in dieser Hinsicht zwar effizienter, gleichzeitig aber bisher nur in der Lage, IA-32 und EM64-t zu virtualisieren und fiel somit aufgrund der Portierbarkeitsanforderung ebenfalls aus der Betrachtung. Des weiteren hat eine Betrachtung des Virtualbox Quellcodes der nicht Hardware-unterstützten Virtualisierungmodule gezeigt, dass Qemu Code für die Binary-Translation Implementation verwendet wird.

Aufgrund der vielen Ausschlusskriterien ist Qemu die einzige geeignete Plattform. Mit der zum Zeitpunkt der Ausarbeitung bereits bestehenden Implementation , genannt PyQemu, und der detaillierten Kenntnisse der Architektur wurde diese als Evaluationsbasis ausgewählt. Eine bereits bestehende Instrumentierungsimplementation, die Qemu ebenfalls als Basis verwendet, ist Temu[39]. Bei näherer Betrachtung der Konzeptionsentscheidungen des Projekts während der Ausarbeitung zeigte sich jedoch, dass eine performante instruktionsbasierte Programmanalyse nur mit massiven Anpassungen möglich gewesen wäre, sodass PyQemu in diesem Fall die besten Voraussetzungen bot.

	PyQemu	**PIN**	**Bochs**	**Temu**	**Valgrind**	**Ether**
Portierbarkeit	✓	×	×	✓	×	×
Performanz	✓	✓	×	×	✓	✓
OS unabhängig	✓	×	✓	✓	×	✓
Emulierte Hardware	✓	×	✓	✓	×	✓

Abbildung 5.1 Vergleich frei verfügbarer Implementationen zur Evaluation der Heuristiken

5.1.4 Beschreibung der Analyseumgebung

Bisher wurden die grundlegenden Entscheidungen bezüglich der Evaluationsplattform erklärt und begründet. Im Folgenden soll die im Kontext der Evaluation genutzte Umgebung beschrieben werden. Dies dient gleichzeitig zur Einleitung in die weiteren Designentscheidungen des Analysesystems. Abbildung 5.2 zeigt die Umgebung des zu analysierenden Programms. In einer beliebigen Softwareumgebung, welche mit Qemu kompatibel ist, findet eine Vollvirtualisierung des Analysesystems statt. Dies bedeutet, dass jegliche vom Gastsystem erreichbare Hardware nicht real existiert, sondern durch die Evaluationssoftware emuliert wird. Die sich daraus ergebenden Vorteile liegen in der Möglichkeit, mithilfe von Softwareanpassungen Zugriffe auf die emulierte Hardware messen zu können. Eine virtuelle Netzwerkkarte kann somit modifiziert werden, um Netzwerkpakete von und an das Gerät mitzuschneiden oder zu verändern. Ein anderer Anwendungsfall wäre das Messen von Zugriffen auf den virtuellen Arbeitsspeicher.

Innerhalb der Umgebung der emulierten Hardware wird im Kontext der Ausarbeitung ein Windows-Betriebssystem ausgeführt. Konzeptuell kann dabei jedes System verwendet werden, das zu einer Prozessor Architektur kompatibel ist, die von Qemu unterstützt wird. Im Kontext der Ausarbeitung fiel die Wahl auf ein Windows XP Betriebssystem, da dieses bei Analysen von Schadsoftware im Gegensatz zu anderen Betriebssystemen aufgrund der hohen Verbreitung unter Anwendern besser geeignet

ist. Die Ausführung des zu analysierenden Programms findet in der Benutzerumgebung des virtualisierten Betriebssystems statt. Bei der Konzeptionierung wurde aufgrund der Gefahren einer Erkennung des Analysesystems durch die ausgeführte Software besonderer Wert auf den Verzicht von Modifikationen der virtualisierten Umgebung gelegt. Jegliche Daten über Stati dieser Umgebung werden deshalb transparent aus dem Wirtsystem heraus abgegriffen. Hierzu gehört beispielsweise die Erkennung laufender Prozesse. Um komfortabel zu einem Ausführungpunkt zurückspringen zu können, wird die von Qemu bereitgestellte Funktionalität zum Anfertigen und Zurücksetzen von Momentaufnahmen genutzt. Diese kann jedoch nicht innerhalb der Analyse einer Instruktionssequenz genutzt werden, sondern dient nur der manuellen Vorkonfiguration der Analyseumgebung, beispielsweise der Installation eines zu analysierenden Programms.

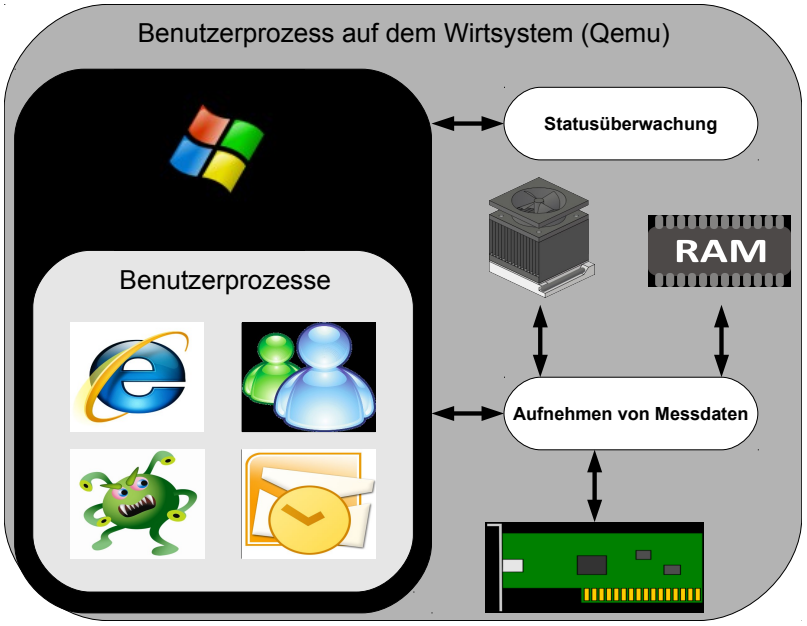

Abbildung 5.2 Umgebung des Analysesystems

5.1.5 Architektur des Evaluationssystems

Die Architektur des Evaluationssystems in Abbildung 5.3 illustriert die verschiedenen Bestandteile der Auswertungssoftware. Mehrere Teile der Software wurden direkt als Anpassungen der Virtualisierungssoftware Qemu implementiert. Dies war notwendig, um bei Prozessen, die während der Ausführung der zu analysierenden Software in vielen Iterationen durchlaufen werden, das Performanzkriterium zu erfüllen. Um gleichzeitig eine möglichst hohe Flexibilität bezüglich zukünftiger Modifikationen zu erreichen, wurden alle anderen Softwaremodule in der Programmiersprache Python entwickelt. Mittels eigener Implementation von Schnittstellen findet ein Einbindungsprozess der in Python geschriebenen Module beim Start der virtuellen Maschine in den Qemu Kontext statt.

Jegliche Datengewinnung über das Verhalten eines analysierten Programms setzt auf die von Qemu eingesetzte Binary-Translation-Technik. Auf dieser basierend wurden

verschiedene sogenannte Hooking-Module entwickelt, welche die Codeübersetzung zur Laufzeit beeinflussen und somit Aufrufe an Ereignisbehandlungsroutinen in die Instruktionssequenz des Programms transparent einbauen. Ein Beispiel wäre das Setzen von Haltepunkten oder das Einbauen von Aufrufen an Funktionen zur Analyse der Speicherzugriffe. Da das Sammeln jeglicher zur Laufzeit verfügbarer Messdaten nicht in einer akzeptablen Laufzeit möglich ist, nutzen verschiedene Aggregatoren die von den Hooking-Modulen zur Verfügung gestellten Programmdaten, um diese möglichst zu komprimieren, ohne dabei einen Informationsverlust bezüglich der Datengrundlage der angewandten Heuristiken in Kauf nehmen zu müssen. Eine einfache und bereits beschriebene Heuristik ist die Messung der Anteile arithmetischer Instruktionen in einem bestimmten Abschnitt der Instruktionssequenz. Dabei könnten die ausgeführten Blöcke auf ein persistentes Medium geschrieben werden, um eine spätere Analyse zu ermöglichen. Eine datenaggregierende Variante, wie sie in diesem Fall implementiert wurde, speichert jedoch nur die Anzahl der Vorkommnisse verschiedener Instruktionsklassen und kann somit die Datenmenge signifikant verringern. Die notwendigen Informationen zur Berechnung der arithmetischen Anteile gehen dabei nicht verloren. Aggregierte Daten werden je nach Komplexität der Verarbeitungsprozesse entweder durch ein Logging der Rohdaten persistent gespeichert oder zur Laufzeit von einem Heuristikmodul in einen repräsentierenden Wert umgewandelt.

Da die Instrumentierungsbereiche zu analysierender Prozesse zur Laufzeit eingeschränkt werden sollen, um kürzere Analysezeiten zu ermöglichen, werden verschiedene Ereignisse, wie beispielsweise das Ausführen eines neuen Prozesses, durch die implementierte Schnittstelle an die in Python entwickelten Überwachungsmodule übergeben. Gleiches geschieht mit den zur Laufzeit von den Heuristiken ermittelten Zwischenwerten, die jeweilige vorher definierte Schwellenwerte überschreiten und somit einer Erkennung entsprechen. Die Grundlage hierzu bildet eine Überwachung des virtualisierten Betriebssystems, um Ereignisse mit den zugehörigen Objekten wie beispielsweise Threads assoziieren zu können. Pro Thread werden hier jeweils die zu analysierenden Codebereiche festgelegt, um an anderer Stelle, beispielsweise bei der Auswertung durch Heuristiken, auf Threadkontextdaten zugreifen zu können. Schlussendlich werden die Ereignisse, die von den in Qemu integrierten Modulen erzeugt wurden, entweder ohne eine weitere Verarbeitung an das Subsystem zur persistenten Speicherung der aggregierten Daten weitergegeben oder zuvor erneut durch zusätzliche Heuristiken verarbeitet, um zur Laufzeit Aussagen zu berechnen. Ein Beispiel für eine solche Heuristik ist eine mögliche Suche nach algorithmentypischen Konstanten im Speicher des zu analysierenden Programms. Da sich eine solche Suche als verhältnismäßig aufwändig herausgestellt hat, ist es notwendig, diese nur beim Auftreten einiger weniger Ereignisse zu starten. Ereignisse, die sich anbieten, wären zum Beispiel das Starten und Beenden von Prozessen. Solche Ereignisse führen daher dazu, dass bereits aggregierte Daten von einer zusätzlichen Heuristik verarbeitet werden und somit Ereignisse wie ein Prozessstart, Ursache künstlicher Ereignisse, zum Beispiel eines Konstantenfunds, sein können.

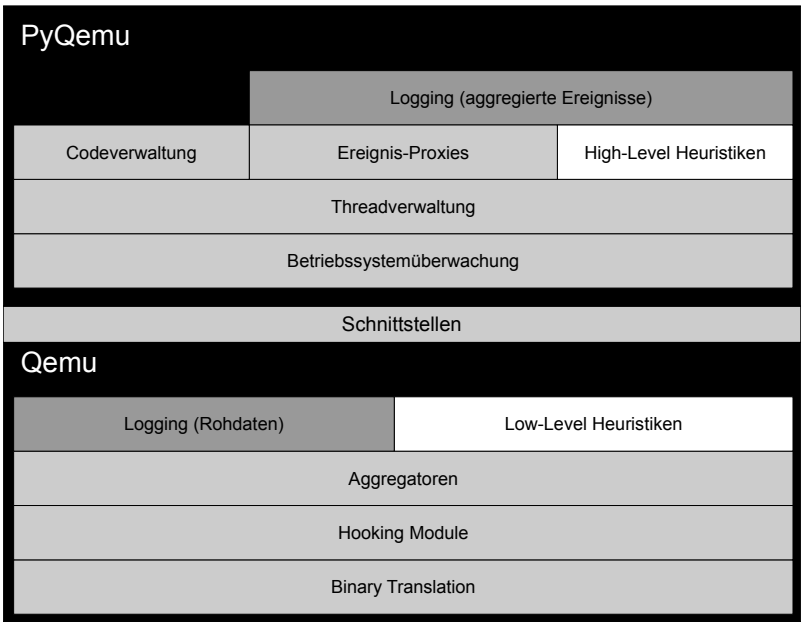

Abbildung 5.3 Softwarestapel des Evaluationssystems. Die Daten werden von unten nach oben verarbeitet.

5.1.6 Verarbeitungsweg der Messdaten

Neben dem Aufbau des Evaluationsystems beeinflusst die Menge der Messdaten, sowie die Art der Verarbeitung die Anwendbarkeit im Hinblick auf Analyseaufwand und Erweiterbarkeit. Im Rahmen der Entwicklung der Analyseplattform musste entschieden werden, zu welchem Zeitpunkt Ereignisse verarbeitet und in eine eventuelle Erkennung überführt werden. Grundsätzlich bestehen hier zwei Möglichkeiten: Eine Variante ist die Speicherung aller notwendigen Daten zur Auswertung nach der Ausführung des zu analysierenden Programms. Der Vorteil ist, dass die Laufzeitkosten der Erkennungsmethoden dabei nicht die reale Laufzeit während des Durchlaufs beeinflussen. Als Nachteil muss dabei aber berücksichtigt werden, dass gegebenenfalls sehr viele Daten zur Laufzeit gespeichert werden müssen und deshalb nicht in allen Fällen eine tatsächliche Laufzeitverbesserung die Folge sein muss, da das Schreiben auf persistente Medien verhältnismäßig langsam ist. Das Gegenstück zu dieser sogenannten a posteriori-Analyse bildet die Aggregierung und Anwendung der Erkennungsmethoden zur Laufzeit des zu analysierenden Programms, wobei sich hier Vor- und Nachteile der zuvor beschriebenen Methode umkehren.

Wichtig ist eine theoretische Gegenüberstellung der Laufzeitkomplexitäten einer Live-Auswertung sowie der a posteriori-Analyse durch eine Speicherung der Rohdaten. Das Ablegen von Daten auf persistente Medien liegt in der Komplexitätsklasse $O(n)$. In der Praxis benötigen Schreibvorgänge jedoch einen hohen konstanten Mehraufwand, der durch den Kontextwechsel zum Betriebssystem und der Initialisierung des Datenaustauschs zwischen den Geräten entsteht. Erkennungsmethoden hingegen haben als untere Schranke meist $\Omega(\lg n \cdot n)$. Dies liegt daran, dass Zugriffe auf gespeicherte Daten in der Praxis nicht schneller als in $\lg n$ durchführtbar sind und eine Iteration über alle gesammelten Datensätze meist notwendig ist. Dafür entsteht bei der Verarbeitung ein wesentlich geringerer konstanter Aufwand. In Fällen, in denen eine Verarbeitung wenige Ereignisse pro Iteration bei einer gleichzeitig hohen Anzahl an Gesamtiterationen aufweist, trägt die Auswertung der Daten zur Laufzeit

zu einer geringeren Ausführungsdauer bei. Werden jedoch wie im Fall von Speicherzugriffen zu bestimmten Zeitpunkten besonders viele Ereignisse innerhalb einer Iteration ausgewertet, sollte die Methode der Speicherung von Rohdaten angewandt werden. Es muss jedoch berücksichtigt werden, dass eine a posteriori-Auswertung keine Ereignisse zur Laufzeit produzieren kann, was unter Umständen aber gewollt ist, um weitere Analysemaßnahmen zu beginnen.

Während der Testphase des Evaluationssystems konnte die Laufzeit durch die zusätzliche Anwendung von a posteriori-Analysen auf ein Zehntel reduziert werden. Weitere Tests zur Speicherung von Rohdaten des ausgeführten Programmcodes hatten jedoch im Gegensatz zu einer Live-Auswertung einen gegenteiligen Effekt. Darin begründet liegt die hybride Anwendung beider Methoden bei der Konzeptionierung der Analysesoftware.

Bisher blieb die Frage unbeantwortet, wie die zur Laufzeit der analysierten Software produzierten Daten gemessen werden. Anhand von Abbildung 5.4 sollen daher die Module erklärt werden, die Programmdaten verarbeiten und aufbereiten. Ereignisse können in zwei Basiskategorien unterteilt werden: Instruktionsdaten und Speicherdaten. Speicherdaten beinhalten jegliche Informationen darüber, was das Programm verarbeitet. Hierzu gehören Registerwerte, globale Variablen, dynamisch allokierte Speicherbereiche, Konstantensegmente und der Stapelspeicher. Instruktionsdaten hingegen geben Aufschluss darüber, wie ein Programm die Speicherdaten verarbeitet. Kurz gefasst bestehen die Instruktionsdaten aus der Instruktionssequenz des Programms bis zum jeweiligen Ausführungspunkt. Instruktionsdaten werden durch Modifikationen des Binary-Translation-Prozesses gesammelt. Dieser unterteilt die Instruktionssequenz eines Programms in einzelne Blöcke, genannt Basic-Block, die einzeln übersetzt und verwaltet werden. Das Ende eines Basic-Block wird dabei hinter das erste Vorkommnis einer Sprunginstruktion gelegt. Dieses grundlegende Konzept ermöglicht es, die Blöcke anhand der Adresse der jeweils ersten Instruktion bis zum Zeitpunkt einer Modifikation des assoziierten Speicherbereichs zu identifizieren und auf logischer Ebene mit Eigenschaften zu verknüpfen. Wichtig dabei ist es zu verstehen, dass durch die Art der Unterteilung zum Zeitpunkt vor der ersten Ausführung der ersten Instruktion eines Basic-Blocks der Programmstatus direkt nach der Ausführung des Blocks ausschließlich von der Spezifikation des Befehlssatzes abhängt. Einzelne Basic Blöcke sind daher mit Ausnahme des Auftretens asynchroner Ereignisse problemlos statisch analysierbar. Eigenschaften, die im Kontext der Ausarbeitung mit Basic-Blöcken verknüpft werden, sind zum Beispiel Zugehörigkeiten zu einer Routine oder die während der Ausführung des jeweiligen Blocks gelesenen Daten. Die Instruktionsanalyse verwaltet diese Daten und generiert gegebenenfalls Ereignisse, die durch Aggregationsmodule verarbeitet werden.

Die Messung der Speicherzugriffe erfolgt ebenfalls durch Eingriffe in den Binary-Translation-Prozess. Dabei werden die Daten mithilfe eines sogenannten Schattenspeichers auf die tatsächliche Menge an relevanten Speicherdaten reduziert, indem aus dem Arbeitsspeicher gelesene Werte zusammen mit der Speicheradresse und der Adresse der zuletzt lesenden Instruktion verwaltet werden. Soll beispielsweise eine Suche nach Algorithmenkonstanten initiiert werden, ist es hilfreich, ausschließlich den Schattenspeicher zu benutzen, da somit die sich im Addressbereich des zu analysierenden Prozesses befindlichen Daten, die bis zum Zeitpunkt der Suche nicht gelesen wurden, unberücksichtigt bleiben und die zugehörige zu lokalisierende Codestelle des letzten Zugriffs direkt ermittelbar ist. So lässt sich die dynamische Analyse

nutzen, um Suchräume zu reduzieren, unnötige falsche Positive zu vermeiden und über die Musteridentifizierung hinaus auch den verarbeitenden Code zu lokalisieren.

Wie bereits beschrieben, werden die gemessenen Daten von den Aggregatoren anschließend soweit notwendig zusammengefasst und persistent gespeichert.

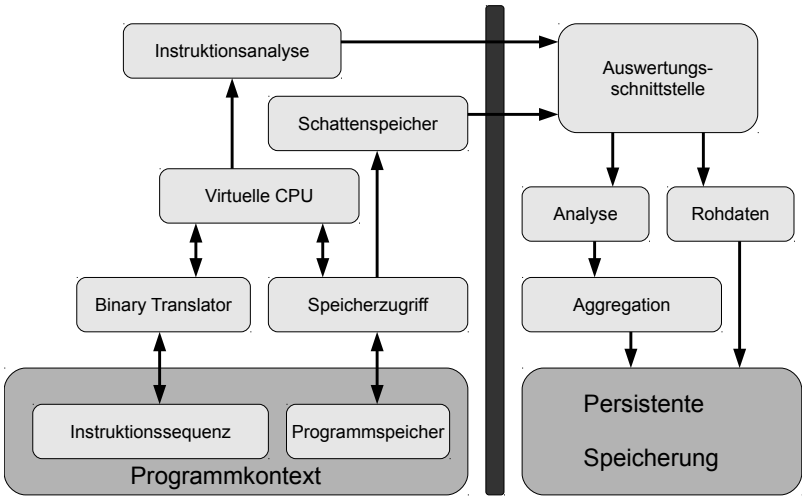

Abbildung 5.4 Verarbeitungswege der gemessenen Daten

5.1.7 Beschreibung der Analyseabläufe

Die Anwendung der Erkennungsmethoden in der Evaluations- und Produktivphase der Analysesoftware kann in drei verschiedene Tätigkeitsbereiche unterteilt werden, die durch Abbildung5.5 illustriert werden:

1. Vorbereitung
2. Ausführung
3. Anschließende Analyse und Verifikation der Ausgaben

Innerhalb der Vorbereitungsphase wird die Umgebung zur Ausführung des jeweils zu analysierenden Programms eingerichtet. Dazu gehört die Installation von Abhängigkeiten sowie das Kopieren an der Ausführung beteiligten Bibliotheksdateien auf das Wirtsystem für den Zugriff während der Analysezeit. Weiterhin muss die Evaluationsplattform dahingehend konfiguriert werden, dass bekannt ist, welche Prozesse in die Messungen mit einbezogen werden und welche Programm- und Bibliotheksdateien dazugehören.

Anschließend wird das zu analysierende Programm nach der erfolgreichen Wiederherstellung der Virtualisierungsumgebung aus der letzten Momentaufnahme gestartet. Der Analyseprozess beginnt mit dem Erreichen des sogenannten Programmeintrittspunktes. Dieser wird in der initial ausgeführten Programmdatei festgelegt und markiert den Zeitpunkt, an dem benötigte Bibliotheken durch den Laufzeitlinker und -Lader in den Addressbereich des Prozesses eingebunden wurden. Prinzipiell kann bereits während des Ladens von Bibliotheksdateien Code in deren Initialisierungsroutinen ausgeführt werden, der für das Verhalten des zu analysierenden Programms relevant ist. Die Einschränkung, erst bei Erreichen des Programmeintrittspunktes zu instrumentieren, ist jedoch nicht konzeptbedingt und wurde ausschließlich aus Zeitersparnisgründen in Kauf genommen. Auf die Evaluationsergebnisse mit legitimen Programmen wird diese Tatsache keinen Einfluss haben. Gleiches gilt mit hoher Wahrscheinlichkeit für die zu testende Schadsoftware, da diese so gewählt wird, dass jeweils immer nur eine Programmdatei existiert und keine Techniken zum Einsatz kommen, bei denen Initialisierungsroutinen von Bibliotheken die Analyseergebnisse verfälschen. Während des Ausführungsverlaufs sammelt das Evaluationssystem die für die Heuristiken benötigten Daten, die neben einer persistenten Speicherung zum Teil direkt zu Ergebnissen verarbeitet werden.

Die Ausführung endet entweder bei der Terminierung des Programms oder durch den Benutzer der Evaluationssoftware. Wird die Ausführung beendet bevor das Programm dies eigenständig veranlasst hat, speichert die Analysesoftware vor der Beendigung der Virtualisierungsumgebung ein Abbild des zu diesem Zeitpunkt im Prozesskontext verwalteten Speichers. Dieses Abbild dient einer späteren statischen Programmanalyse. Da während der Laufzeit bereits einige Erkennungsmethoden ausgeführt wurden, liegen zu diesem Zeitpunkt bereits Teile der Analyseergebnisse in Form einer Protokolldatei vor.

Die dritte und letzte Phase beginnt mit der Ausführung der a posteriori-Analyse mithilfe der zur Laufzeit des zu analysierenden Programms gespeicherten Ereignisse. Der Programmverlauf wird soweit notwendig rekonstruiert und in eine weitere Protokolldatei der angewendeten Heuristiken überführt. Danach liegen alle Ergebnisse der Erkennungsmethoden vor. Während der Evaluation sowie später im Produktiveinsatz ist es hilfreich, ein Programm zur statischen Analyse zu nutzen. Im Kontext der Ausarbeitung wird der Interactive Disassembler (IDA)[9] genutzt. Hierzu wurden Erweiterungen implementiert, mit deren Hilfe die von den Erkennungsmethoden produzierten Ausgaben in IDA angezeigt werden können. Zusätzlich sind die betroffenen Codestellen mit Kommentaren annotiert, deren Inhalt beispielsweise aus den berechneten Ausgabewerten der Heuristiken bestehen. Die zur Laufzeit des zu analysierenden Programms erstellten Speicherabbilder können mithilfe anderer implementierter Erweiterungen rekonstruiert und für eine statische Betrachtung durch einen Analysten vorbereitet werden. Dies hilft bei der Evaluation der Erkennungsmethoden im Fall von Schadsoftware, da diese oftmals gepackt und verschleiert ist, sodass ansonsten gegebenenfalls viel Zeit für die Rekonstruktion der originalen Programmdateien verloren gegangen wäre. Da die meiste Schadsoftware zu Beginn der Ausführung im Speicher komplett entpackt wird, ist diese Methode für die Verifikation der Ergebnisse ausreichend.

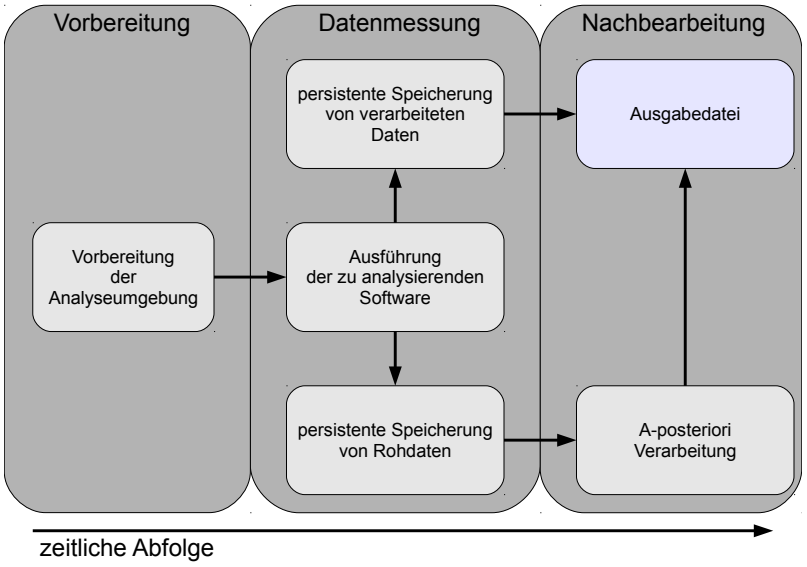

Abbildung 5.5 Zeitliche Einordnung beteiligter Prozesse zur Erkennung von Kryptographie

5.2 Lösungsansätze der Erkennungsprobleme

Im Kapitel "Problembeschreibung" wurde das Problem der Erkennung von Kryptographie in Software näher beschrieben und eingegrenzt. Im Folgenden soll die Aufteilung aus Definition 5 aufgegriffen werden, um heuristische Lösungsansätze, die in dieser Ausarbeitung evaluiert werden, zu kategorisieren und somit die Aussagekraft im Fall einer erfolgreichen Erkennung bestimmen und vergleichen zu können.

5.2.1 Allgemeines Erkennungsproblem

Das allgemeine Erkennungsproblem, wie es in dieser Ausarbeitung definiert wurde, klassifiziert eine Teilinstruktionssequenz entsprechend dem implementierten Algorithmus entweder als kryptographisch oder nichtkryptographisch. Benutzt werden in diesem Zusammenhang die Methoden der Instruktionsanalyse sowie Entropiemessungen der verarbeiteten Daten.

Wie bereits Wang festgestellt hat, nutzen Instruktionssequenzen kryptographischer Algorithmen eine zu anderen Algorithmen vergleichsweise hohe Anzahl an arithmetischen- sowie bitbasierten Prozessorinstruktionen[42]. Dabei wurde diese Eigenschaft in Form des kumulativen Anteils ab einem festgelegten Zeitpunkt, wie beispielsweise dem Auslesen von Daten aus einer Netzwerkverbindung, implementiert. Die beschriebene Methode basiert allerdings auf sehr starken Annahmen bezüglich der Verarbeitungsreihenfolge eingehender Daten des zu analysierenden Programms. Caballero entwarf aus diesem Grund eine von dieser Annahme unabhängige Methode, die den Anteil der vermeintlich oft vorkommenden Instruktionen in Funktionen ab einer Größe von 20 Instruktionen berechnete[5]. Der Schwellenwert, bei dem eine Funktion von Interesse markiert wurde, lag bei 55 Prozent. Gröbert übernahm diesen Wert in seiner Masterarbeit, übersah dabei jedoch, dass Caballero nicht ausschließlich kryptographische Algorithmen im Sinne seiner Definition, sondern zusätzlich

auch Dekodierungsfunktionen erkennen wollte und wendete die Heuristik anstatt auf Funktionen auf einzelne Blöcke an. Untermauern lässt sich diese Problematik mit Abbildung 5.6, in der mehrere Messungen der Anteile arithmetischer Operationen in Blöcken von jeweils 1000 Instruktionen der Ausführungssequenz illustriert werden. Sowohl das Programm Notepad als auch die curl-HTTP-Abfrage enthielt soweit verifizierbar keine kryptographischen Routinen. Aescrypt hingegen verschlüsselte während des Durchlaufs eine Datei mit dem Rijndael Algorithmus. Curl nutzte in der HTTPS-Abfrage mehrere kryptographische Algorithmen in einer Komposition. Zum einen bestätigt die Grafik zumindest die von Wang beschriebene Tendenz. Zum anderen ist aber auch sichbar, dass ein Schwellenwert der Methode nach Caballero allgemein zu niedrig liegt, da der durchschnittlich gemessene Anteil bei der curl-HTTP-Abfrage zum Teil sehr nah an die 55 Prozent Marke herankommt und zum Ende sogar kurz übersteigt. Hiermit ist eine erneute Ermittlung des Schwellenwerts für die Anwendung der Methode nach Caballero innerhalb dieser Ausarbeitung gerechtfertigt. Wie bei Gröbert werden im Kontext dieser Ausarbeitung ebenfalls nur einzelne Instruktionsblöcke betrachtet. Der Grund dafür liegt in der allgemeinen Schwierigkeit, das Ende einer Funktion zu erkennen. In Fällen in denen eine Heuristik, wenn auch mit veränderten Schwellenwerten, sowohl auf Funktionen als auch auf Instruktionsblöcke angewendet werden kann, ist letzteres aufgrund der sicheren Trennungsmöglichkeit von Blöcken vorzuziehen.

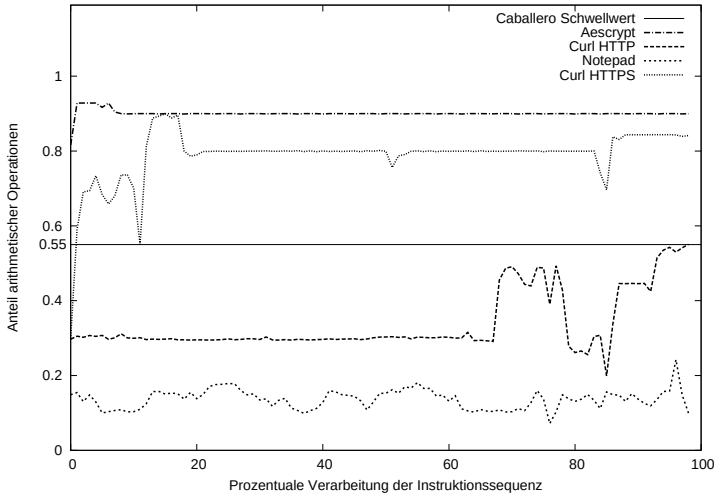

Abbildung 5.6 Vergleichsmessung der Anteile arithmetischer und bitbasierter Operationen

Eine weiterer Aspekt, der für alle dem Autor bekannten kryptographischen Algorithmen gleichsam gilt, ist die Eigenschaft der hohen Entropie in Chriffretexten. Dies liegt daran, dass die Form verschlüsselter Daten möglichst schwierig vorherzusagen sein muss und resultiert daher in einer Vergleichbarkeit mit Pseudozufälligen Datenströmen. Lutz verwendete in seiner Arbeit die sogenannte skalierte Entropie über ein Tupel von Datenbytes b mit der Länge i[26]:

$$H(b) = \frac{-\sum_{i=1}^{256} \frac{|b|_i}{n} \cdot log_2 \frac{|b|_i}{n}}{log_2(min(n,256))}$$

$|b|_i$ ist die Anzahl der Vorkommnisse des i. Bytes in b. Der Quotient sorgt dabei für eine Vergleichbarkeit von Tupeln verschiedener Längen trotz der Rechnung mit einem konstanten Alphabet der 256 möglichen Werte eines Bytes.

Lutz berechnete mit dieser Formel die Entropiedifferenz verarbeiteter Daten in Schleifendurchläufen, da kryptographische Algorithmen wie Blockchiffren oftmals in mehreren Runden ver- und entschlüsseln. Eine Betrachtung mehrerer Implementationen symmetrischer Chiffren sowie Hashfunktionen ergab jedoch, dass dies nicht der Fall sein muss. Viele der vorhandenen Referenzimplementationen nutzen eine Kette von Makros, um mit möglichst wenig Sprungbefehlen zu effizienterem Code vom Compiler umgesetzt zu werden. Entropiemessungen wurden aus diesem Grund jeweils am Ende einer Routine und nicht mithilfe einer Schleifenerkennung implementiert, was darüber hinaus auch durch die zeitlichen Bearbeitungsgrenzen begründet ist. Während Lutz ausschließlich die Entropiedifferenz gelesender und geschriebener Werte in der Verarbeitung betrachtete, wurden vom Autor ebenfalls zwei weitere mögliche Methoden abgeleitet:

1. Entropiemessung der gelesenen Werte
2. Entropiemessung der geschriebenen Werte

Die Idee, die beiden Methoden zusätzlich zur Anwendung zu bringen, basiert darauf, dass kryptographische Algorithmen in Modi verwendet werden, in denen der eigentliche Entschlüsselungsprozess keine Entropieveränderung zur Folge haben muss. Beim sogenannten Cipher-Block-Chaining (CBC) in Abbildung 5.7 wird der erste Datenblock mit einem Initialisierungsvektor und jeder weitere vor der Verschlüsselung mit dem letzten verschlüsselten Block Bitweise XOR-Verknüpft. Damit wird verhindert, dass zwei gleiche Klartextblöcke immer auch auf den gleichen Chiffretext abbilden, da dies ein Sicherheitsproblem darstellen würde. Ähnliche Methoden werden genutzt, um Blockchiffren in der selben Art und Weise wie Stromchiffren verwenden zu können. Offensichtlich hätte die Messung der Entropiedifferenz zur Folge, dass man nicht die genutzte kryptographische Routine sondern die davor bzw. dahinterliegende XOR-Verknüpfung erkennen würde. Für Lutz' Anwendungsfall der automatisierten Extrahierung der Klartextkommunikation mag dies ausreichend sein. Im Kontext dieser Ausarbeitung ist die Tatsache jedoch ein erhebliches Problem, da Blockchiffren in der Praxis so gut wie immer im CBC oder einem ähnlichen Modus verwendet werden. Aus diesem Grund werden jeweils Entropiemessungen der gelesenen sowie der geschriebenen Daten getätigt und diese mit absoluten Schwellwerten beurteilt.

Es existieren einige allgemeine Probleme bei der auf Entropiemessungen basierenden Erkennungsmethoden. Auch Komprimierungsverfahren erzeugen eine hohe Ausgabeentropie, die von der Entropie verschlüsselter Texte meist kaum zu unterscheiden ist. Zusätzlich gelten hohe Entropiewerte der gelesenen Daten mit hoher Wahrscheinlichkeit für jegliche Routinen, die innerhalb der Programmausführung auf verschlüsselte Daten zugreifen oder diese kopieren. Insgesamt kann also gesagt werden, dass Entropiemessungen der verarbeiteten Daten für sich betrachtet keine guten Aussagen liefern können, da zu viele falsche Positive entstehen werden. Dennoch wurde

die Erkennungsart in die Ausarbeitung mit aufgenommen, da sie gegebenenfalls in Kombination mit der angepassten Caballero-Heuristik als reine Ausschlussheuristik eine Verbesserung erwirken kann. In der Evaluation wird darüber hinaus geprüft, ob aus der Anwendung eines kryptographischen Algorithmus folgt, dass eine hohe Entropie verarbeiteter Daten existiert. Wäre dies der Fall, können Raten falscher Positive aller restlichen Erkennungsmethoden gegebenenfalls verkleinert werden.

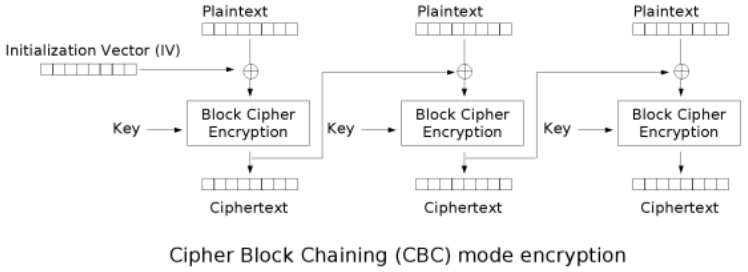

Abbildung 5.7 CBC Verschlüsselung (Lizenz: Creative Commons)

5.2.2 Klassenspezifisches Erkennungsproblem

Im Gegensatz zum allgemeinen Erkennungsproblem wird durch die Definition des Klassenspezifischen Erkennungsproblems die Menge kryptographischer Algorithmen in disjunkte Teilmengen zerlegt. Bei der Problemdefinition selbst wurde jedoch noch nicht festgelegt, welche Klassen im Kontext der Ausarbeitung erkannt werden sollen und durch welche Eigenschaften sie sich voneinander vermeintlich unterscheiden. Folgende Klassen kryptographischer Algorithmen sollen durch die hier beschriebenen Methoden unterschieden werden:

1. Asymmetrische Kryptoalgorithmen
2. Symmetrische Kryptoalgorithmen und kryptographische Hashfunktionen

Asymmetrische Kryptoalgorithmen rechnen mit Ausnahme der Nutzung elliptischer Kurven weitestgehend mit Restklassenringen und verwenden deshalb die Modulo-, Multiplikations- und Divisionsoperation mit Ganzzahlen. Da zur sicheren Benutzung Zahlen in Größenordnungen verwendet werden, die außerhalb der Domäne von 32-Bit breiten Registern liegen[2], werden die Operationen in einzelnen Teilschritten auf kleine Bereiche der Operanden aufgeteilt, welche der von der Architektur zur Verfügung gestellten Bitbreite entsprechen. Aus diesem Grund ist davon auszugehen, dass die Instruktionssequenz einer Implementation einer Modulo Operation in kryptographischen Algorithmen mehrere entsprechende Instruktionen beinhalten wird. Eine von der Caballero Methode abgeleitete Heuristik ist daher die Messung des prozentualen Anteils der Instruktionen mul (Multiplikation) und div (Division und Modulo). Existierende Befehlssatzerweiterungen, welche die genannten Operationen teilweise mit erweiterter Bitbreite durchführen, werden im Kontext dieser

Ausarbeitung nicht berücksichtigt. Sollte eine Erkennung während der Evaluation fehlschlagen und der Grund die Nutzung einer solchen Erweiterung in der Instruktionssequenz sein, wird dies explizit erwähnt.

Symmetrische Kryptoalgorithmen und kryptographische Hashfunktionen sind zwar bezüglich ihrer Anwendung eher in zwei verschiedenen Klassen einzuordnen, allerdings gelten die im Kontext der Ausarbeitung gemessenen Eigeschaften nach vorhergegangenen Tests für beide Mengen gleichermaßen, sodass diese hier als eine gemeinsame Klasse betrachtet werden müssen. Zwei verschiedene Methoden wurden ausgearbeitet, um eine Erkennung zu realisieren. Eine Methode zur Erkennung der Klasse asymmetrischer Kryptoalgorithmen arbeitet Äquivalent zur Caballero-Heuristik, mithilfe von Messungen der Anteile klassentypischer Instruktionen. Dazu zählen Schiebe-, Und- und Oderverknüpfungs- sowie Rotationsoperationen und die XOR-Operation. Da gegenüber der allgemeinen Erkennung kryptographischer Algorithmen wesentlich weniger Instruktionsarten berücksichtigt werden, muss der Schwellenwert voraussichtlich niedriger sein.

Eine weitere, hier zu evaluierende Methode, basiert auf einer Analyse des Datenflusses einer Routine und wird im Folgenden erläutert. Betrachtet man sowohl bei symmetrischen Chiffren als auch bei kryptographischen Hashfunktionen die Unterschiede der Ausgaben bei kleinen Modifikationen der Eingaben, so lässt sich feststellen, dass diese sehr groß sind. Konfusion und Diffusion sind die ursächlichen Eigeschaften, die für eine hohe Sicherheit der Algorithmen notwendig sind und sich deshalb auch in allen modernen Algorithmen dieser Klassen wiederfinden lassen[43]. Bezüglich der Datenflüsse lässt sich daher folgern, dass ein Byte der Eingabe einen großen Teil der Ausgabe beeinflusst. Um diese Eigenschaft messen zu können, wird in drei Schritten vorgegangen:

1. Modellierung der Beeinflussung durch einen Graph
2. Aufteilung des Graphen in Teilgraphen
3. Berechnung des Teilgraph mit maximaler Beeinflussung

Als Grundlage dient, wie bei den auf Entropiemessungen basierenden Methoden, die Einteilung der Instruktionssequenz in Routinen. Innerhalb einer Routine werden Zugriffe auf den Arbeitsspeicher gemessen und in Lese- sowie Schreibzyklen unterteilt. Ein Lesezyklus wird durch eine Schreiboperation beendet. Analog gilt dies für Schreibzyklen. Während der Zyklen werden die Adressen, auf die durch die Instruktionen zugegriffen werden, gespeichert und bis zum Abschluss des nächsten Zyklus vorgehalten. Es existiert für jede Ausführung einer Routine ein gerichteter Graph, der zu Beginn leer ist. Wird ein Schreibzyklus von einer Leseoperation beendet, so werden alle Adressen, auf die im vorhergehenden Lesezyklus zugegriffen wurde, in den Routinengraph als Knoten hinzugefügt, falls sie noch nicht existieren. Gleiches geschieht mit allen Adressen des soeben beendeten Schreibzyklus. Anschließend wird dem Graph von jedem eine Leseadresse repräsentierenden Knoten, eine Kante zu allen Schreibadressen repräsentierenden Knoten erstellt. Unter der Annahme, dass zwischen der Verarbeitung eines zu ver- oder entschlüsselnden Bytes keine weiteren Speicheroperationen stattfinden, repräsentiert der Routinengraph im Groben die Beeinflussung der Speicheradressen untereinander. Ein Beispielergebnis des Vorgangs wird in Abbildung 5.8 illustriert.

Am Ende einer Routinenausführung ist somit ein Graph vorhanden, aus dem in den folgenden Schritten ein Teilgraph $G(V, E)$ mit maximalem $\frac{|E|}{|V|}$ ermittelt werden soll. Um dieses Problem mit möglichst wenig Rechenaufwand lösen zu können, wurde die Annahme genutzt, dass Daten, auf denen Konfusion und Diffusion angewendet wird, mit sehr hoher Wahrscheinlichkeit als Chiffre- bzw. Klartext in einem zusammenhängenden Speicherbereich liegen. Eine Heuristik wertet für jeden Knoten v die Kantenanzahl zu Nachbarknoten $e := |\{(v, v')|(v, v') \in E \wedge |v - v'| \leq d\}|$ mit einer maximalen Speicherdistanz d aus, wobei die Differenzoperation hier eine Differenz der repräsentierten Adresswerte meint. Abhängig von diesem Parameter und einem minimal zu akzeptierenden e ergibt sich eine neue Menge von Knoten, welche immer noch Speicheradressen repräsentieren. Nun kann ein einfacher Algorithmus zusammenhängende Speicherblöcke extrahieren, welche anschließend als Teilgraphen aufgefasst werden, sofern die jeweils darin enthaltene Knotenanzahl den angegebenen Blocklängenparameter l nicht unterschreitet. Abbildung 1 beschreibt die Methode in Form von Pseudocode. Prinzipiell könnten hier auch Taint-Tracking-Algorithmen angewendet werden, um eine bessere Genauigkeit bzgl. des Beeinflussungsgraphen zu erzielen[7]. Zum einen wäre dies jedoch sehr ineffizient, da es in der Praxis mehrere Quellen - Bytes im Eingabepuffer - und Senken - Bytes im Ausgabepuffer - gibt. Zum anderen kann durch die Annahme, dass die zu verarbeitenden Puffer im Speicher in zusammenhängenden Blöcken liegen, die durch die nur rudimentär implementierte Datenflussmessungsmethode entstehenden Ungenauigkeiten durch den Algorithmus wieder herausgefiltert werden.

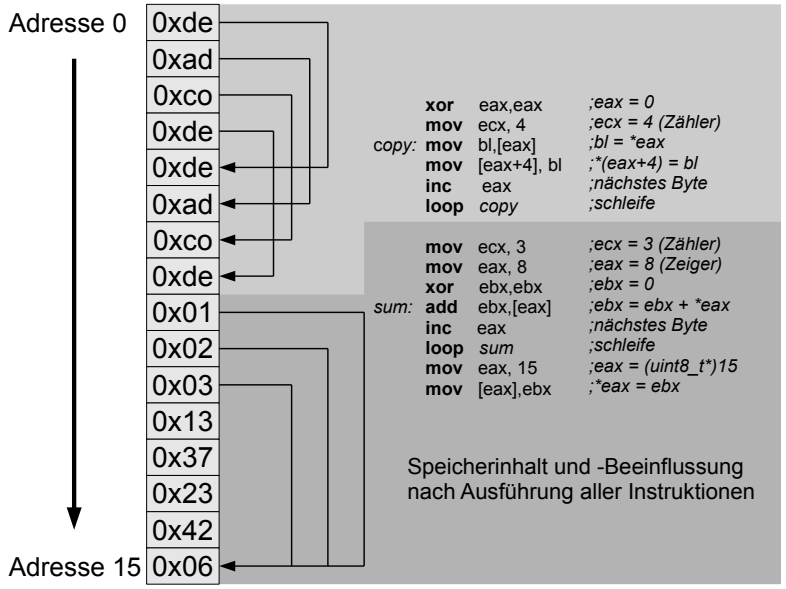

Abbildung 5.8 Illustration des aus einer Instruktionssequenz entstehenden Flussgraphen

Require: Beeinflussungsgraph $G(V, E)$
Ensure: Block B mit größter innerer Beeinflussung
d := Suchweite zur Einteilung in Blöcke
e := Anzahl benötigter Kanten in Suchweite
l := Minimale Blockgröße
V := Speicheradressen repräsentierende Knoten
E := Beeinflussungskanten
$V' := \{v | \exists x_1, ..., x_e (\wedge_{i=1}^{e} (\wedge_{j=1}^{e} (j \neq i \rightarrow x_j \neq x_i) \wedge x_i \in V \wedge v \in V \wedge |x_i - v| \leq d \wedge (v, x_i) \in E))\}$
for all zusammenhängende Speicherblöcke b aus V' **do**
if $|b| \geq l$ und $EB := \{(v_1, v_2) | v_1 \in b, v_2 \in b, (v_1, v_2) \in E\}$ und $\frac{|EB|}{|b|}$ maximal **then**
B := b
end if
end for

Algorithm 1 Berechne Block mit größter innerer Beeinflussung

5.2.3 Algorithmenspezifisches Erkennungsproblem

Die beschriebenen Lösungsansätze des klassenspezifischen Erkennungsproblems sind nicht in der Lage, einzelne Algorithmen zu identifizieren, da in den zu erkennenden Klassen offensichtlich jeweils mehrere Algorithmen enthalten sind. Wie in der Problembeschreibung erklärt, unterliegen Lösungen des algorithmenspezifischen Erkennungsproblems zwar der Limitierung auf die Erkennung bekannter Algorithmen, können dafür jedoch den genutzten Algorithmus im Falle einer erfolgreichen Erkennung auch identifizieren. Als verwandte Arbeiten wurden bereits Werkzeuge vorgestellt, die basierend auf einer statischen Analyse versuchen, Konstanten und Instruktionssequenzen mittels einer Mustererkennung zu finden. Instruktionssequenzen fallen im Kontext der Ausarbeitung jedoch unter das Implementierungsspezifische Erkennungsproblem, da sie vor allem durch die Implementierung und nicht durch den zugrundeliegenden Algorithmus festgelegt werden. Konstanten hingegen können als Eigenschaft eines Algorithmus akzeptiert werden. Zwar ist es möglich, diese zum Beispiel für den symmetrischen Kryptoalgorithmus Rijndael zu verändern, allerdings wurden bestimmte Teile dieser Konstanten, in diesem Kontext S-Boxen genannt, bei der Standardisierung des Algorithmus als Advanced Encryption Standard (AES) festgelegt. Eine Implementierung mit modifizierten S-Boxen wäre darüber hinaus nicht kompatibel zu anderen Implementierungen, die dem Standard folgen. Ähnlich verhält es sich bei vielen anderen kryptographischen Algorithmen, sodass die Suche nach Konstanten als implementationsunabhängige Methode betrachtet werden kann. Da die ausgewählte Evaluationsplattform mittels dynamischer Analyse arbeitet, soll diese ebenfalls zur Konstantenerkennung genutzt werden. Hieraus ergeben sich zwei Vorteile: Zum einen sinkt die Rate falscher Positive in Fällen rapide ab, in denen das Programm eine komplette Bibliothek von Kryptoalgorithmen nutzt, da eine statische Analyse hier alles unabhängig von der tatsächlichen Nutzung erkennen würde. Zum anderen existieren Implementationen, in denen Konstanten erst zur Laufzeit zusammengerechnet oder entschleiert werden. Sofern der Grad der Verschleierung also das Ablegen der Konstanten im Speicher zulässt und die Werte zwischen zwei Erkennungssuchläufen nicht wieder überschrieben wurden, stellt die Verschleierung

und Zusammensetzung zur Laufzeit im Gegensatz zu einer statischen Analyse kein Problem dar. Zur Evaluation wurden Konstanten aus einer Testmenge von Algorithmen extrahiert:

1. Blowfish
2. Camellia
3. Cast
4. DES
5. GOST
6. IDEA
7. MD5
8. RC5
9. Rijndael
10. RIPEMD-160
11. Serpent
12. SHA1
13. SHA2
14. SHA256
15. SHA384
16. SHA512
17. TIGER
18. Twofish
19. Whirlpool

Verschiedene Implementationen dieser Algorithmen werden zum Test der Erkennungsrate der Konstantensuchmethode während der Evaluation betrachtet. Trotz der Behauptung, dass kryptographische Algorithmen existieren, für die eine Menge von Konstanten jeweils fest definiert ist, ist es unklar, ob sich diese Konstanten in der Praxis auch unabhängig von implementierungsspezifischen Faktoren finden lassen. Ein asymmetrisches Kryptosystem ist unter den gegebenen Algorithmen nicht vertreten. Dies liegt vor allem daran, dass diese Art kryptographischer Algorithmen ohne für einen Suchdurchlauf verwertbare Konstanten arbeiten. Wie jedoch die zitierte Arbeit von Klein[19] zeigt, lassen sich Muster aus der standardisierten Darstellungsweise der Schlüssel extrahieren und erkennen. Der Autor hat diese Methode bereits außerhalb der Ausarbeitung nachimplementiert und erfolgreich getestet[27]. Aus diesem Grund wird die Methode hier nicht extra evaluiert.

5.2.4 Implementierungsspezifisches Erkennungsproblem

Lösungen des implementierungsspezifischen Erkennungsproblems erkennen Instruktionssequenzen, die einer festen Implementierung eines Algorithmus entsprechen. Im Gegensatz zu Lösungen des algorithmenspezifischen Erkennungsproblems sind sie auf deutlich kleinere Mengen von Instruktionssequenzen beschränkt. Algorithmen werden im Kontext dieser Ausarbeitung anhand von Konstanten erkannt. Diese Methode ist in Fällen problematisch, in denen die zu einem Algorithmus gehörenden Konstanten nicht vorhanden oder nicht eindeutig genug sind, um diesen identifizieren zu können. Die Stromchiffre RC4 ist ein solches Beispiel. Mittels eines Schlüsselvorbereitungsalgorithmus wird ein Schlüssel vor der Nutzung modifiziert. Anschließend dient ein pseudozufälliger Zahlengenerator zur Erstellung eines Schlüsselstroms, der byteweise auf den Klartext mit der XOR-Operation verrechnet wird. Hierzu werden keinerlei eindeutige Konstanten verwendet, was den Lösungsansatz des algorithmenspezifischen Erkennungsproblems scheitern lässt.

Eine von Gröbert[10] abgeleitete Methode zur Lösung des implementationsspezifischen Erkennungsproblems ist die Erstellung von Signaturen aus Instruktionen. Verschiedene Instruktionssequenzen entstehen aus einer Implementation durch unterschiedliche Vorgehensweisen bei der Umwandlung z.B. durch verschiedene Compiler. Gröbert bildete daher Schnittmengen verschiedener Instruktionssequenzen, die sich beim Umwandeln einer Implementation durch die Nutzung verschiedener Compiler

und deren Optimierungsstufen ergaben. Eine Evaluation dieser Methode kann nicht zusammen mit den Lösungsansätzen der restlichen Erkennungsprobleme geschehen, da hier im Unterschied zu allen anderen Methoden die Frage nach einer Unterscheidbarkeit verschiedener Implementationen gleicher Algorithmen im Hauptfokus steht. Des weiteren hat die Erkennung spezieller Implementierungen, wie bereits in Kapitel 4 erläutert, nur einen kleinen Anwendungsbereich, beispielsweise beim Erkennen von Referenzcode, der Fehler enthält und ein paar Algorithmen, die ähnliche Eigenschaften wie RC4 aufweisen. Aus diesen Gründen werden Lösungen des implementierungsspezifischen Erkennungsproblems im Kontext dieser Ausarbeitung nicht evaluiert. Stattdessen wird auf die Evaluation von Gröbert[10] verwiesen.

Eine spezielle Menge von Instruktionssequenzen, die Implementationen kryptographischer Algorithmen entsprechen, besteht aus den von der Windows-Krypto-API[30] zur Verfügung gestellten Funktionen. Da dies die einfachste und offensichtlichste Form zur Integration von Kryptographie in ein Programm unter Windows darstellt und die Möglichkeit zur Erkennung von API-Aufrufen innerhalb des Evaluationssystems bereits aus anderen Gründen zur Verfügung stand, wurde diese genutzt. Hierzu werden beim Laden der zugehörigen Bibliotheken die Funktionen der Windows Krypto API lokalisiert und entsprechende Haltepunkte gesetzt. Werden diese Funktionen aufgerufen, speichert das System die Codestellen, von denen aus die Aufrufe erfolgten.

6

Implementierung

Kapitel 5 beschreibt das Design der Analyseplattform. In diesem Teil sollen darüber hinaus einige Implementationsdetails der Qemu-Anpassungen beschrieben werden. Sie bestehen zum einen aus der Modifikation der Qemu-Binary-Translation-Implementation und zum anderen aus der Extrahierung von Windows-internen Datenstrukturen zur Betriebssystem- und Prozessüberwachung.

6.1 Anpassungen des Virtualisierungssystems

Zur Messung der für die Heuristiken benötigten Daten mussten einige Anpassungen an Qemu durchgeführt werden. Bei Instruktionsemulatoren wie Bochs reicht es, die für die relevanten Instruktionen und Speicherzugriffe verantwortlichen Codeteile durch das Hinzufügen von Methodenaufrufen in eigene Auswertungsfunktionen zu modifizieren. Dass Qemu sogenanntes Binary-Translation nutzt, gestaltet die Implementation schwieriger, bietet allerdings auch einige Vorteile. Einzelne Basic Blöcke, wie sie in Kapitel 5 erläutert werden, durchlaufen vor der Ausführung einen Übersetzungsprozess, in dem jede einzelne Instruktion sequentiell abgearbeitet wird. Zu diesem Zeitpunkt können ebenfalls Methodenaufrufe eingebaut werden, die in Qemu-internen Code springen. Dies wurde ursprünglich zur Emulation einzelner komplexer Instruktionen, wie die Verarbeitung von Gleitkommazahlen, implementiert. Über reine Methodenaufrufe hinaus lässt sich zu diesem Zeitpunkt auch Speicher allokieren, der im Kontext des übersetzten Basic-Blocks gültig bleibt. Zur Laufzeit kann dann bei dessen Ausführung durch die Instrumentierungsroutine auf diesen blocklokalen Speicher zugegriffen werden. Hierdurch lassen sich die Codeteile der virtuellen Maschine mit Informationen annotieren, die zum Zeitpunkt zur Auswertung nicht separat ermittelt werden müssen. Die Implementation der Qemu-Binary-Translation bietet also gute Voraussetzungen für eine instruktionsbasierte dynamische Programmanalyse.

Basic-Blöcke werden von Qemu in einem Cache gehalten. Hierdurch kann die Laufzeit bei mehrmaliger Ausführung gleicher Codeteile verringert werden, da bereits

übersetzte Blöcke direkt angesprungen werden können. Jegliche Datenerhebung im Kontext der Evaluationsplattform basiert auf den soeben vorgestellten Konzepten des Binary-Translation. Im Folgenden werden die Implementationen der einzelnen Ereignisquellen vorgestellt.

6.1.1 Haltepunkte

Haltepunkte sind in Qemu bereits in der Implementation eines GDB Stub[40] vorhanden. Damit ist die Verwendung eines externen Debuggers möglich, der sich transparent an die virtuelle Maschine anhängen kann. Die angebotenen Schnittstellen reichten für die Anforderungen des Evaluationssystems nicht aus, da die Haltepunkte im Kontext der Ausarbeitung mit dem jeweils laufenden Prozess assoziiert sein müssen und die Implementation in Qemu durch verkettete Listen in Tests zu inperformant war.

Die Haltepunkte des Evaluationssystems wurden durch Nutzung von AVL-Bäumen in zwei kaskadierten Ebenen implementiert. Ein erster Baum beinhaltet die Prozessobjekte, die einen Zeiger auf die jeweiligen Haltepunkte enthalten. Diese werden separat in einem jeweiligen Baum verwaltet, in dem die Haltepunkte in den Knoten zusätzlich durch eine doppelt verkettete Liste verbunden sind. Zum Zeitpunkt der Übersetzung eines Codeblocks kann somit direkt der am nächsten liegende Haltepunkt ausgewählt werden. Dies gestaltet die Suche in den Datenstrukturen pro Block effizienter, da diese in den meisten Fällen nur ein einziges mal durchgeführt werden muss und in den folgenden Iterationen über die Instruktionen eines Blocks ein einfacher Vergleich ausreichend ist.

6.1.2 Instruktionshooking

Die verschiedenen Instruktionen, die im Kontext der Evaluationsplattform Ereignisse generieren, wurden bei der Implementation in verschiedene Klassen unterteilt, um eine granularere Instrumentierung ermöglichen zu können. Eine sogenannte Instrumentierungskonfiguration, welche die Ereignisgenerierenden Klassen festlegt, wird für jeden Prozess einzeln verwaltet und zum Zeitpunkt der Blockübersetzung genutzt um zu entscheiden, zu welchen Instruktionen Aufrufe zu Auswertungsmethoden hinzugefügt werden müssen. Zur Laufzeitverkürzung kann diese Möglichkeit in Zukunft genutzt werden, um beispielsweise Routinen, die durch die Heuristiken als in Frage kommende kryptographische Routinen bereits ausgeschlossen werden konnten, ebenfalls aus dem Instrumentierungsprozess auszuschließen.

6.1.3 Speicherzugriffshooking

Um eine Datengrundlage für speicherbasierte Erkennungsmethoden zu schaffen, müssen die Zugriffe auf den virtuellen Arbeitsspeicher des zu analysierenden Programms abgegriffen werden. Der Binary-Translation-Teil in Qemu stellt Funktionen bereit, welche Code für den virtualisierten Speicherzugriff generieren. Dies ist notwendig,

da die Adressen der virtuellen Maschine in die tatsächlichen Adressen des Qemu-Prozesses umgerechnet werden müssen. Die Funktionalität ist dahingehend angepasst, dass zu jeder Lade- oder Speicheroperation instrumentierter Prozesse ein entsprechender Aufruf der Auswertungsmethoden für Speicherzugriffe zum Zeitpunkt der Befehlsübersetzung hinzugefügt wird.

6.1.4 Implementation der Mustersuche

Zu den Erkennungsmethoden zählen unter anderem eine Konstanten- sowie eine Instruktionsmustersuche. Erreicht der zu analysierende Prozess seinen durch die ausgeführte Datei festgelegten Eintrittspunkt, werden die Muster in das jeweils zuständige Modul hinzugefügt. Während die Instruktionsmustersuche nach Beendigung einer Routine über die Liste der in dessen Kontext ausgeführten Codeblöcken durchgeführt wird, muss die Konstantensuche explizit aufgerufen werden. Aufrufe erfolgen bei der Ausführung von Haltepunkten bekannter Windows-API-Funktionen wie dem Starten bzw. Beenden von Prozessen und Threads. Da die Implementation der Suche aufgrund der prinzipiell großen Anzahl von Mustern skalieren sollte, wurde der String Matching Algorithmus von Aho und Corasick[1] verwendet. Dieser baut einen Zustandsautomaten zur parallelen Erkennung der Muster auf und kann einen Suchstring somit in linearer Zeit verarbeiten. Wird ein Muster gefunden, generiert das verantwortliche Modul ein entsprechendes Ereignis, welches dann vom Python-Logger gespeichert wird. Eine Aufbereitung in Form von Annotationen des gefundenen Suchmusters mit der gehörigen Beschreibung vereinfacht die Interpretation durch den Analysten.

6.1.5 A posteriori-Auswertung

Da Erkennungsmethoden, wie die Graph- und Entropieauswertung, zu hohe Laufzeitkomplexitäten besitzen, um die Ereignisse zur Ausführungszeit der zu analysierenden Software verarbeiten zu können, wurde ein Typ-Wert ähnliches Protokoll zur Serialisierung der benötigten Daten implementiert. Für jeden instrumentierten Thread wird dabei eine zugehörige Logdatei erstellt um die Ereignisse persistent zu speichern. Zur a posteriori-Analyse existieren mehrere in Python geschriebene Klassen, welche die Daten wieder auslesen und die Ereignisse rekonstruieren können. Somit kann die Ausführung des analysierten Programms im Nachhinein zumindest beschränkt auf die gespeicherten Ereignisklassen beliebig oft abgespielt werden. Die Erkennungsmethoden werden während dieses Prozesses angewendet.

6.1.6 Datenaggregation

Die Aggregation gemessener Daten ist ein wichtiger Bestandteil der Konzeptionierung des Evaluationssystems. Die Implementation wurde durch verschiedene Module realisiert, welche für sich gesehen nur ein Teilproblem lösen und jeweils eine oder mehrere Datenquellen sowie -senken besitzen. Die Datenquellen der Module auf der untersten Ebene sind die bei der Blockübersetzung eingebauten Aufrufe in die Auswertungsroutinen. Alle darüberliegenden Module registrieren sich per Callback bei

den jeweiligen Quellmodulen und bieten anderen Modulen selbst wieder eine API zur Registrierung von Callbacks. Somit ist die Auswertung der Ereignisse sehr modular, sodass Heuristiken mit ähnlichen Informationsbedürfnissen, wie bereits implementierte, sehr schnell integrierbar sind. Ein weiterer Vorteil liegt in der Aggregation und der damit verbundenen Komprimierung auftretender Ereignisse. Während verschiedener Tests zeigte sich ein deutlicher Performanzgewinn. Dies erklärt sich durch die sinkende Menge an Daten, welche persistent gespeichert werden müssen. Zusätzlich zu den Aggregations- und Verarbeitungsmodulen existieren weiterhin Verwaltungsmodule, die Kontextinformationen für die Auswertungsvorgänge zur Laufzeit bereitstellen. Hierzu gehört beispielsweise die separate Speicherung übersetzter Basic-Blöcke mit zusätzlichen Informationen wie den prozentualen Anteilen bestimmter Instruktionen oder auch das Prozessverwaltungsmodul, welches einer Heuristik prozessbezogenen Speicherplatz bietet, um Ereignisse im richtigen Kontext verarbeiten zu können. Diese Architektur würde es bei Multiprozessorsystemen zusätzlich erlauben, die Module in einzelnen Threads zu implementieren und damit gegebenenfalls kürzere Laufzeiten zu erreichen. Dies wurde bisher nicht getestet.

6.2 Betriebssystem- und Prozessüberwachung

Da Qemu aufgrund der Unabhängigkeit der in der virtuellen Maschine ausgeführten Software keine Statusinformationen ausliest, diese aber zur Instrumentierung unbedingt notwendig sind, musste diese Möglichkeit hinzugefügt werden. Zum einen ist es wichtig, aktuelle Prozessinformationen der momentan aktiven Prozesse zu erhalten, um auftretende Ereignisse dem richtigen Prozesskontext zuordnen zu können. Zum anderen müssen Adressen bestimmter Codebereiche bekannt sein, um beispielsweise Funktionsadressen aufzulösen, wenn Haltepunkte gesetzt werden sollen.

6.2.1 Prozess- und Threaderkennung

Ist die Analyseplattform nicht in der Lage, zwischen ausgeführten Prozessen der Virtualisierungsumgebung zu unterscheiden, können auftretende Ereignisse den jeweiligen Prozessen nicht zugeordnet werden. IA32-kompatible Prozessoren besitzen sogenannte Kontrollregister, von denen eines (CR3) einen Zeiger auf Speicherverwaltungsinformationen des aktuellen Prozesses beinhaltet. Wird vom Taskplaner ein neuer Prozess zur Ausführung ausgewählt, modifiziert das Betriebssystem den in CR3 enthaltenen Zeiger und stellt den vorher gesichterten Prozesskontext wieder her. Da Qemu die Hardware, im für die Evaluationsplattform gewählten Virtualisierungsmodus, vollständig emuliert, kann eine Modifikation des Registers abgegriffen werden. Zu solchen Zeitpunkten können dann prozessinterne Datenstrukturen ausgelesen werden, um Informationen, wie den Prozessnamen oder geladene Bibliotheken, zu extrahieren. Solange der Prozess aktiv ist, bleibt dieser Kontext erhalten, sodass auftretende Ereignisse korrekt zugeordnet werden können. Die Definition der Datenstrukturen in Python wurden aus der Implementation von Pandoras-Bochs von Lutz Böhne[4] entnommen und an die von Qemu angebotenen Schnittstellen angepasst. Im Detail funktioniert das Extrahieren der Prozessdaten durch Auslesen eines Registers, das unter Windows auf eine Speicherstelle zeigt, an der sich

der sogenannte Process-Environment-Block (PEB) befindet. Dieser wiederum zeigt auf weitere Datenstrukturen, die Informationen über den aktiven Prozess, wie zum Beispiel den Prozessnamen, enthalten. Um Kontextwechsel zwischen Threads innerhalb eines Prozesses erkennen zu können, reicht die gerade beschriebene Methode jedoch nicht aus. Da Threads innerhalb eines Prozesses im gleichen Adressraum existieren, muss das CR3 Register nicht immer neu geladen werden. Ein einfach zu implementierender Ansatz ist die Unterscheidung von Threads anhand spezieller Stapelspeicherregister. Da jeder Thread einen eigenen Stapelspeicher besitzt, kann dies als Merkmal innerhalb eines Prozesses verwendet werden. In Zukunft wäre es ein besserer Ansatz, die Scheduling-Routinen im Kernel als Haltepunkte zu setzen und somit Kontextwechsel robuster zu erkennen.

6.2.2 Lokalisierung geladener Bibliotheksfunktionen

Die Lokalisierung geladener Bibliotheken, Funktionen und sonstiger Codeteile ist wichtig, um den Instrumentierungsbereich einschränken zu können. Sobald ein Prozess mit Namen bekannt ist und sich in der Liste der durch die Konfiguration der Evaluationsplattform spezifizierten zu analysierenden Programme befindet, wird ein Haltepunkt auf den Programmeintrittspunkt festgelegt. Aus Gründen der Effizienz und der einfachen Implementierung befinden sich die zu analysierenden Programme und Systembibliotheken zusätzlich im Dateisystem des Wirtsystems, sodass die Analyseplattform viele Informationen statisch aus den ausführbaren Dateien auslesen kann. Das Herausfinden der Startadressen geladener Objekte geschieht jedoch zwangsläufig durch Auslesen des Adressraums eines laufenden Prozesses, da die Informationen aus den ausführbaren Dateien nicht ausreichen. Die statischen Informationen werden dann anhand dieser Startadressen in die entsprechenden virtuellen Adressen umgerechnet. Somit erhält die Evaluationsplattform Adressdaten geladener Bibliotheken und der jeweils exportierten Funktionen, sodass beispielsweise Haltepunkte bestimmter Windows-API-Funktionen korrekt gesetzt werden können.

6.3 Limitierungen

Aufgrund der zeitlichen Begrenzung konnten einige implementationsspezifische Beschränkungen des Evaluationssystem nicht mehr aufgehoben werden. Im Folgenden werden diese Limitierungen genauer erläutert.

6.3.1 Limitierungen aufgrund des Translation Caches

Der Cache, der bereis übersetzte Blöcke vorhält, wird Translation-Cache genannt. Eine genauere Erklärung der Funktionsweise ist notwendig zum Verständis einiger Instrumentierungseinschränkungen. Qemu selbst wertet keinerlei interne Zustände der in der virtuellen Maschine ausgeführten Software aus, um unnötige Abhängigkeiten zu vermeiden. Aus diesem Grund werden übersetzte Codeteile verschiedener Prozesse zusammen in einem einzigen Translation-Cache verwaltet. Problematisch ist dies bei der Instrumentierung von Bibliotheken, die im Adressraum verschiedener

Prozesse existieren, da die Codeteile dann für alle Prozesse instrumentiert sind und zur Laufzeit gegebenenfalls zu viele Ereignisse verarbeitet werden müssten. Eine Lösung wäre die Leerung des Translation-Caches nach jedem Prozesskontextwechsel. Während der Implementation wurde dies getestet und führte zu einer inakzeptablen Laufzeitverlängerung. Ein weiterer Lösungsweg ist die Erweiterung von Qemu zur Verwaltung von separaten Translation-Caches pro Prozess. Aus Zeitgründen war dies jedoch bis zur Durchführung der Evaluation nicht möglich, da die Anpassungen mit einem erheblichen Aufwand verbunden wären. Der gewählte Lösungsweg besteht darin, dass geladene Bibliotheken der analysierten Software anhand einer Whitelist instrumentiert werden, wobei Systembibliotheken, die auch von anderen Prozessen genutzt werden, unberücksichtigt bleiben. Die Evaluationsergebnisse werden hierdurch allerdings nicht verfälscht, da die Funktionalität dieser Bibliotheken weitestgehend dokumentiert und bekannt ist. Relevante Codeteile gehören zur Software selbst und sind trotz der Einschränkungen komplett instrumentierbar. Da es für eine breitere Anwendung des Evaluationssystems erforderlich sein kann, auch Codebereiche zu instrumentieren, die von vielen Prozessen genutzt werden, ist die Implementation prozessspezifischer Translation-Caches in Zukunft sinnvoll.

6.3.2 Limitierungen der Evaluationsumgebung

Die Betriebssystemüberwachung der Evaluationsumgebung umfasst momentan ausschließlich Windows XP. Somit können beispielsweise Programme unter Linux noch nicht analysiert werden. Weiterhin existieren nur Anpassungen an der Binary-Translation-Implementation des IA32-Befehlssatzes, sodass Prozessor-Architekturen wie ARM nicht instrumentierbar sind. Mit Ausnahme von Temu, das mehrere Betriebssysteme in der virtuellen Maschine unterstützt, besitzen alle anderen erwähnten Implementationen zur dynamischen Analyse analoge Limitierungen. Im Kontext von Schadsoftware sind sie weitestgehend akzeptabel, da ein Großteil der aktuell sich im Umlauf befindenden Schadprogramme immer noch Windows XP Systeme zum Ziel haben. In Zukunft wird zumindest die Unterstützung von x86-64 erforderlich sein.

7

Evaluation

In den vorigen Kapiteln wurde die Problemstellung genauer analysiert und einige heuristische Verfahren zu deren Lösung vorgestellt. Wie erfolgreich diese sind, soll in diesem Teil der Ausarbeitung evaluiert werden. Dies geschieht vor allem im Hinblick auf mögliche Kombinationen von Erkennungsmethoden zur effizienteren und effektiveren Lösung der Erkennung von Kryptographie in Software. Im Folgenden wird die Herangehensweise genauer erläutert und begründet. Im Anschluss werden jeweils die einzelnen Evaluationsabschnitte, deren Ergebnisse und Interpretationen beschrieben.

7.1 Herangehensweise

Kapitel 4 zeigt, dass die Erkennung von Kryptographie in Software ein komplexes Problem ist und dass dieses Problem auf verschiedenen Ebenen betrachtet werden muss. Die Zielsetzung der Evaluation ist zum einen, dass die Erkennungsmethoden auf ihre praktische Einsetzbarkeit hin überprüft werden. Zum anderen soll gezeigt werden, ob Kombinationen von Heuristiken die Erkennungsleistung insgesamt oder für bestimmte Mengen verbessern kann. Dabei wird in drei Schritten vorgegangen: In der ersten Phase werden Programme getestet, die von ihrer Funktionalität her genauestens bekannt sind. Diese Programme enthalten kryptographische Algorithmen, deren Ausführung zur Laufzeit sichergestellt ist. Eine gute Testabdeckung vorhandener kryptographischer Algorithmen kann nur so erreicht werden, da die Nutzung von Kryptographie in Software weitestgehend nur auf einer kleinen Menge dieser Algorithmen beruht. Darüber hinaus ist es mithilfe von für die Evaluation entworfenen Programmen einfacher möglich, mit anderen oft verwendeten Algorithmen, beispielsweise zum Sortieren, die verwendeten Methoden auf falsche Positive zu testen. Die erste Phase kann daher als algorithmenbasierte Evaluation betrachtet werden. Eine zweite Phase, in der mit bekannten Programmen gearbeitet wird, die nicht zwangsweise im Quellcode vorliegen, soll die praktische Anwendbarkeit des Gesamtsystems evaluiert werden. Hierzu wird einerseits Software ausgewählt, die kryptogra-

phische Algorithmen nutzt, um die Erkennungsraten festzustellen. Zusätzlich dient eine Testmenge von Software ohne Kryptographie dazu, die Anzahl an falschen Positiven zu testen. Diese Trennung gilt der Vorbäugung von möglichen Beeinflussungen durch kryptographische Algorithmen auf andere Codestellen. Falsche Positive bei Programmen mit Kryptographie werden zusätzlich betrachtet. In der dritten und letzten Phase wird erneut die Motivation des Einleitungskapitels aufgegriffen, die sich unter anderem auf das Reverse-Engineering von Schadsoftware bezieht. Hierzu dient eine aktuell verbreitete Schadsoftware, für die versucht wird, mithilfe der Erkennungsmethoden und dem Evaluationssystem möglichst viele Informationen über die Nutzung kryptographischer Algorithmen zu erhalten. Die Evaluation einer Programmausführung besteht jeweils aus mehreren Schritten. Im ersten Schritt wird die zu analysierende Software in der virtuellen Maschine installiert. Anschließend sorgt die Instrumentierung während der Ausführung dafür, dass Daten über den Programmablauf gewonnen werden. In einer a posteriori-Analyse werden die zur Laufzeit gespeicherten Ereignisdaten zu Aussagen der Heuristiken verarbeitet und in eine Ergebnisdatei geschrieben. Mithilfe dieser Ergebnisdatei prüft der Autor die jeweiligen Ausgaben durch eine statische Analyse des Programmcodes.

7.2 Parametrisierung

Im Folgenden werden die Parameter der Erkennungsmethoden erläutert, die vor der Evaluation auf Basis von Tests während der Implementation festegelegt wurden und für die gesamte Evaluation gültig sind. Als Testmenge dienten einige Referenzimplementationen von Blockchiffren, weshalb die Blockchiffren in der Evaluation aus einer anderen Quelle stammen[33]. Aus Zeitgründen wurden Tests mit asymmetrischen Kryptosystemen und kryptographischen Hashalgorithmen unterlassen:

- Die Caballero-Heuristik in Kapitel 5.2.1 meldet Codeblöcke ab einer Größe von 20 Instruktionen und einem Anteil von mindestens 70 Prozent arithmetischer oder bitbasierter Instruktionen. Der Parameter ergibt sich aus der Beobachtung, dass der Initialwert von 55 Prozent nicht ausreichend war, siehe hierzu Abbildung 5.6, und die relevanten Codestellen während der Implementierungstest den gesetzten Anteil als untere Grenze enthielten. Die 20 Instruktionen wurden aus der Arbeit von Caballero ohne Änderung übernommen.

- Die instruktionsbasierte Erkennung von Blockchiffren und Checksummen, die in 5.2.2 beschrieben wurde, meldet Blöcke mit mindestens 20 Instruktionen, von denen mindestens 40 Prozent in der Menge xor, shift, and, or oder rotation liegen müssen. Da in dieser Heuristik nur ein Teil der Instruktionsmenge enthalten ist, die von der Caballero-Heuristik gemessen werden, musste der Schwellenwert herabgesetzt werden. Die 40 Prozent sind das Minimum der Instruktionsanteile kryptographischer Codestellen, die in der Testmenge während der Implementation gefunden werden konnten.

- Die instruktionsbasierte Erkennung von asymmetrischen Kryptoalgorithmen aus 5.2.2 meldet Blöcke mit mindestens 10 Instruktionen, von denen mindestens 50 Prozent in der Menge mul, div oder add liegen müssen. Die Parameter wurden so angepasst, dass während der Tests nur eine akzeptable Anzahl an

falsche Positive auftraten. Dies war notwendig, da keine Tests mit asymmetrischen Kryptosystemen im Vorfeld durchgeführt wurden. Die minimale Blocklänge ist auf 10 reduziert, da kürzere Blöcke in den Instruktionssequenzen von Bignum-Operationen erwartet wurden. Diese Erwartung basiert auf der gesammelten Erfahrung durch statische Analysen einer Bibliothek zur Anwendung von Arithmetik auf Zahlen, die außerhalb der Domäne der Prozessorregister des Zielsystems liegen[11].

- Die Konstantensuche unter 5.2.3 meldet einen Algorithmus, sobald eine darin vorkommende Konstante gefunden wurde. Die Konstanten wurden aus Referenzimplementationen der jeweiligen Algorithmen entnommen.
- Die Entropiedifferenz aus Kapitel 5.2.1 meldet eine Funktion, sobald die Entropie der gelesenen oder der geschriebenen Werte $0,6$ übersteigt und die Entropiedifferenz größer oder gleich $0,2$ ist. Diese Differenz war notwendig, um die Blockchiffren der Tests restlos erkennen zu können. Der Schwellenwert der Mindestentropie von $0,6$ ergab sich aus der Notwendigkeit zur Reduktion von falschen Positiven.
- Die absolute Entropiemessung, beschrieben in 5.2.1, meldet eine Funktion, sobald die Entropie der gelesenen oder der geschriebenen Werte 0.7 übersteigt. Wie bei den Parametern für die Entropiedifferenzmethode, zeigte sich bei diesen Parametern eine sehr gute Erkennungsleistung bei gleichzeitig vertretbaren Anzahlen falscher Positive. Ein höherer Schwellenwert führte zu falschen Negativen.
- Die Graphheuristik, beschrieben unter 5.2.2, meldet eine Funktion, sobald ein Block gefunden wurde, der mindestens 4 Bytes groß ist und mindestens 8 Kanten in sich selbst besitzt. Die Größen orientieren sich Hauptsächlich an Werten, die zur zuverlässigen Erkennung des Rijndael Algorithmus mindestens notwendig waren. Dieser diente aufgrund seiner Verarbeitungseigenschaften innerhalb der einzelnen Runden als Referenzalgorithmus.
- Aus der Hooking-Funktionalität der Microsoft-Krypto-API unter 5.2.4 werden ausschließlich Aufrufe betrachtet, die eine direkte Ver- oder Entschlüsselungsfunktionalität durchführen.

Aus Gründen der Vollständigkeit wurde die Hooking-Funktionalität der Microsoft-Krypto-API in der Evaluation betrachtet. Es wird jedoch davon ausgegangen, dass diese in den Testfällen wenig Relevanz besitzt, da Verschlüsselungsbibliotheken einen wesentlich größeren Funktionsumfang bieten und gegebenenfalls portabel sind.

7.3 Evaluation mit einzelnen Algorithmen

Die erste Phase der Evaluation dient einer repräsentativen Abdeckung verschiedener Algorithmen. Die vier verwendeten Zeichen zur Beschreibung des jeweiligen Ergebnisses sind in Tabelle 7.1 erläutert.

Die Einteilung ist dadurch begründet, dass die zu erkennenden Mengen von Algorithmen für die Heuristiken unterschiedlich sind. Die Suche nach Konstanten kann

Zeichen	Bedeutung	Beschreibung
$+w$	richtige Positive	Codestellen wurden erfolgreich erkannt
$+f$	falsche Positive	Codestellen hätten nicht angezeigt werden sollen
$-w$	richtige Negative	Codestellen wurden korrekterweise nicht angezeigt
$-f$	falsche Negative	Codestellen hätten angezeigt werden sollen

Abbildung 7.1 Erklärung der Ergebnisbezeichnungen

beispielsweise keine Algorithmen erkennen, für die keine Konstanten zusammengestellt wurden. Werden solche Algorithmen getestet, ist eine Nichterkennung also kein Fehler der Heuristik und wird somit als richtiger Negativer bewertet. Zusätzlich existieren kryptographische Algorithmen, die sich nicht in der Menge der zu erkenndenden Algorithmen aller Heuristiken befinden. Erkennt die Graphheuristik zur Erkennung von Blockchiffren beispielsweise RSA, müsste dies als Falschmeldung interpretiert werden, da RSA nicht in der Menge der zu erkennenden Algorithmen liegt. Die Auswertung in Phase 1 bezieht sich ausschließlich auf Routinen, die zur Kernfunktionalität des zu testenden Algorithmus gehören. Falsche Positive außerhalb dieser Codestellen werden erst in späteren Phasen evaluiert um die Ergebnisse nicht zu überladen.

7.3.1 Test nicht-kryptographischer Algorithmen

Zu Beginn der Evaluation wurden mehrere Programme zur Ausführung von oft genutzten Algorithmen zusammengestellt, um die Robustheit der Erkennungsmethoden zu prüfen. Tabelle 7.2 zeigt die Ergebnisse. Da die verwendeten Testdaten eine hohe Entropie besaßen, weist die Heuristik der absoluten Entropiemessung analog eine hohe Rate falscher Positive auf. Dies zeigt, dass hohe Entropiewerte auch in nichtkryptographischen Algorithmen vorkommen können. Die Falscherkennung der Konstantensuche ist darauf zurückzuführen, dass einige der gesammelten Konstanten, wie in diesem Fall $0x20000000$, oft im Speicher eines Programms vorhanden sind und deshalb in einem Fall zu einem falschen Positiven geführt haben. Aufgrund der Menge der gesammelten Muster war es nicht möglich, diese manuell auf eine Verwendbarkeit hin zu prüfen. In praktischen Szenarien würden solche einfachen Muster jedoch nicht genutzt. Die restlichen Erkennungsmethoden weisen keine falschen Positive auf.

7.3.2 Erkennung symmetrischer Verschlüsselungsalgorithmen

Zur Evaluation der Erkennungsleistung symmetrischer Verschlüsselungsalgorithmen wurden 16 verschiedene Implementationen aus dem Steganographie-Programm OpenPuff[33] extrahiert. Diese eigneten sich besonders gut, da in dem Projekt ausschließlich Modifikationen der Referenzimplementationen verwendet wurden, die an eine generische Schnittstelle zur Ver- und Entschlüsselung gebunden sind. Dies minimierte den Anpassungsaufwand des Quellcodes für die Evaluation. Damit der jeweilige Algorithmus als erkannt in der Tabelle eingetragen werden konnte, musste mindestens eine Unterroutine der Ver- und Entschlüsselungsfunktionen durch die

Programm	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking
Kopieren	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Bubblesort	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Mergesort	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Quicksort	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Heapsort	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
CRC32 Checksummen	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Luhn Checksumme	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
Matrixmultiplikation	$-w$	$-w$	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$
DES	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$	$-w$	$-w$

Abbildung 7.2 Heuristiktest

jeweilige Heuristik angezeigt werden. Der untere, abgetrennte Teil der Tabelle, listet Algorithmen, welche von mindestens einer Heuristik fälschlicherweise erkannt wurden. Die absolute Entropiemessung erkannte alle Funktionen korrekt. Mit 83 Prozent hat die Konstantensuche gemessen an den zu erkennenden Algorithmen die zweit höchste Rate richtiger Positive. Die Caballero-Heuristik erreicht wie auch die Methode zur Erkennung von symmetrischen Verschlüsselungsalgorithmen eine Erkennungsrate von 81 Prozent. Die Taintgraph-Heuristik besitzt eine schlechtere Erkennungsrate mit 68 Prozent. Die Methode der Entropiedifferenzmessung erkannte mit 43 Prozent weniger als die hälfte der zu erkennenden Algorithmen. Relevante API Aufrufe wurden von den Programmen nicht getätigt, weshalb die API Heuristik nichts angezeigt hat. Bei der Konstantensuche fällt auf, dass ein Algorithmus nicht erkannt wurde, obwohl die betreffenden Konstanten im Speicher gesucht wurden. Nach einer Prüfung hat sich herausgestellt, dass dies an der großen Länge der Suchmuster von 16 Bytes gelegen hat. Da während der Durchführung des Algorithmus nur ein Teil der Konstanten genutzt wurde, existierte kein Suchmuster als Ganzes im Schattenspeicher. Die Begründung hierfür liegt in der Tatsache, dass die Programme in den Tests nur verhältnismäßig wenig Daten verschlüsselten. Da die praktische Anwendbarkeit der Heuristiken jedoch in den folgenden Phasen getestet wird, stellt dies kein Problem für die Evaluation als solche dar. Weiterhin hat die Konstantenheuristik mehrere Algorithmen angezeigt, die in den Programmen nicht enthalten waren. Im Fall von RC5 wurde die Erklärung gefunden, dass die Konstantenschnittmengen von RC5 und RC6 nicht leer sind. Die restlichen zwei fälschlicherweise erkannten Algorithmen beruhen auf einer zufälligen Nutzung der Konstanten und wurden der Vollständigkeit halber im unteren abgetrennten Bereich der Tabelle angezeigt. Bei der Prüfung der Konstantenmethode fiel direkt auf, dass bei den richtigen Positiven wesentlich mehr Konstanten des jeweiligen Algorithmus gefunden wurden. Daraus lässt sich ableiten, dass die Einführung eines Schwellenwerts für die Anzahl gefundener Konstanten sinnvoll sein kann und in Zukunft getestet werden sollte. Die

Heuristik zur Erkennung asymmetrischer Kryptosysteme zeigte keine der relevanten Codestellen als kryptographisch an. Dies ist völlig korrekt, da die getesteten Algorithmen allesamt symmetrische Kryptoalgorithmen sind und deshalb nicht in der Menge der zu erkennenden Algorithmen für diese Heuristik enthalten sind.

Algorithmus	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking
Anubis	$+w$	$+w$	$-w$	$-w$	$+w$	$+w$	$+w$	$-w$
Camellia	$+w$	$+w$	$-w$	$+w$	$+w$	$+w$	$+w$	$-w$
Cast256	$+w$	$+w$	$-w$	$+w$	$-f$	$+w$	$+w$	$-w$
Clefia	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$-f$	$-w$
Frog	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$-f$	$-w$
Hierocrypt3	$-f$	$-f$	$-w$	$-w$	$+w$	$+w$	$-f$	$-w$
Idea 128	$+w$	$+w$	$-w$	$+w$	$+w$	$+w$	$+w$	$-w$
Mars	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$+w$	$-w$
RC6	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$+w$	$-w$
Rijndael	$+w$	$+w$	$-w$	$+w$	$-f$	$+w$	$+w$	$-w$
Saferp	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$+w$	$-w$
SC2000	$-f$	$-f$	$-w$	$-w$	$+w$	$+w$	$+w$	$-w$
Serpent	$+w$	$+w$	$-w$	$-f$	$-f$	$+w$	$+w$	$-w$
Twofish	$+w$	$+w$	$-w$	$+w$	$+w$	$+w$	$+w$	$-w$
Cipherunicorn-A	$+w$	$+w$	$-w$	$-w$	$+w$	$+w$	$-f$	$-w$
rc4	$-f$	$-f$	$-w$	$-w$	$-f$	$+w$	$-f$	$-w$
RIPEMD-160	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$	$-w$	$-w$
SHA1	$-w$	$-w$	$-w$	$+f$	$-w$	$-w$	$-w$	$-w$
RC5	$-w$	$-w$	$-w$	$+f$(RC6)	$-w$	$-w$	$-w$	$-w$

Abbildung 7.3 Heuristiktest mit symmetrischen Kryptoalgorithmen

7.3.3 Erkennung asymmetrischer Verschlüsselungsalgorithmen

Eine weitere Klasse kryptographischer Algorithmen, die im Grundlagenkapitel vorgestellt wurden, bilden die asymmetrischen Kryptosysteme. Zum Testen wurden Referenzimplementationen genutzt, die auf der Website von Bruce Schneier gelistet werden[37]. Die Praxis zeigte, dass das Problem der Erkennung dieser Klasse auf eine Erkennung von Funktionen zurückzuführen ist, die arithmetische Operationen auf Zahlen anwenden, die von ihrer Größe her nicht durch die Register heutiger Prozessoren darstellbar sind. Die Nutzung solcher Funktionen ist allerdings nicht ausschließlich an kryptographische Anwendungen gebunden, sodass eine für die Algorithmenklasse eindeutige Menge von Eigenschaften nicht gefunden werden konnte. Zur Evaluation wurden drei oft genutzte Algorithmen herangezogen. Auffällig

ist die gute Erkennung beider Entropieheuristiken mit jeweils 100 Prozent, die auf die Verarbeitung von Schlüsseln mit eine mindestlänge von 1024 Bit zurückzuführen ist. Schlüssel weisen generell eine hohe Entropie auf und könnten laut Prüfung der errechneten Zwischenwerte selbst bei höheren Schwellenwerten noch erkannt werden. Die Heuristik zur Erkennung asymmetrischer Kryptosysteme hat in zwei von drei Fällen die kryptographischen Routinen erfolgreich erkannt. Überraschend ist die schlechte Erkennung der Caballero-Heuristik mit 33 Prozent sowie die Anzahl falscher Positive der Methode zur Erkennung von symmetrischen Kryptoalgorithmen und kryptographischen Hashfunktionen mit 66 Prozent. Die Konstantensuche zeigte wie erwartet keine der asymmetrischen Kryptosysteme an. Die Algorithmen der unteren Tabelle waren in den Programmen enthalten und dienten zur Signaturberechnung während der Ausführung. Eine Evaluation mit weiteren Algorithmen der zu testenden Klasse hätte keine maßgebliche Verbesserung der Repräsentanz bewirkt, da alle Algorithmen auf einer fast identischen Codebasis aufsetzen. Aus diesem Grund wurde an dieser Stelle Zeit für die späteren Evaluationsphasen gespart. Als neue Kernerkenntnis sollte hier betrachtet werden, dass gegebenenfalls hohe Schwellenwerte entropiebasierter Erkennungsmethoden zur Unterscheidung von asymmetrischen Kryptosystemen und Blockchiffren eingesetzt werden können. Die Taintgraph Heuristik erkannte fälschlicherweise ein Kryptosystem. Da keine API Aufrufe getätigt wurden, hat die betreffende Heuristik nichts angezeigt.

Algorithmus	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking
RSA	$-f$	$-w$	$-f$	$-w$	$+w$	$+w$	$-w$	$-w$
Elgamal	$+w$	$+f$	$+w$	$-w$	$+w$	$+w$	$+f$	$-w$
DSA	$-f$	$+f$	$+w$	$-w$	$+w$	$+w$	$-w$	$-w$
RSA-MD5-Hash	$+w$	$+w$	$+f$	$+w$	$+w$	$+w$	$-f$	$-w$
DSA-CAST-Signatur	$+w$	$+w$	$+f$	$+w$	$+w$	$+w$	$+w$	$-w$

Abbildung 7.4 Heuristiktests mit asymmetrischen Kryptosystemen

7.3.4 Erkennung von kryptographischen Hashfunktionen

Zur Sicherstellung der Integritätseigenschaft und zur Authentifizierung kommen überwiegend kryptographische Hashfunktionen zum Einsatz. Um die Erkennungsleistung der Heuristiken bezüglich dieser Klasse zu testen, wurden ebenfalls mehrere Referenzimplementationen aus der Sammlung von Bruce Schneier herangezogen[37].

Die Caballero-Heuristik und die Methode zur Erkennung von symmetrischen Kryptoalgorithmen und kryptographischen Hashfunktionen als auch die absolute Entropiemessung weisen eine Rate richtiger Positive von 100 Prozent auf. Die Nichterkennung des Gost-Algorithmus durch die Konstantensuche ist wie bei den Tests der

symmetrischen Verschlüsselungsalgorithmen auf eine zu große Länge der einzelnen Konstantenmuster zurückzuführen. Die Konstantensuche erkannte trotzdem 83 Prozent der Algorithmen. Die falsche Anzeige des DES Algorithmus ist erneut auf die Nutzung nicht geeigneter Muster zurückzuführen. Da keine Microsoft Krypto API aufrufe getätigt wurden, bleibt die API Heuristik weiterhin ohne richtige Positive. Die Graphheuristik und die Methode zur Erkennung von Entropiedifferenzen zeigen ähnliche Ergebnisse mit einer Rate richtiger Positive von 55 Prozent bzw. 44 Prozent. Die Methode zur Erkennung asymmetrischer Kryptosysteme zeigt korrekterweise keine der Codestellen als kryptographisch an. Ein Rückblick auf die absolute Entropiemessung zeigt, dass bisher alle relevanten Routinen erkannt wurden. In wie weit sich diese Heuristik von einer Referenzheuristik unterscheidet, die alle Funktionen als kryptographisch markiert, zeigen die Tests auf falsche Positive in der nächsten Phase.

Algorithmus	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking
Gost	$+w$	$+w$	$-w$	$-f$	$+w$	$+w$	$-f$	$-w$
Haval	$+w$	$+w$	$-w$	$-w$	$+w$	$+w$	$+w$	$-w$
MD-5	$+w$	$+w$	$-w$	$+w$	$-f$	$+w$	$+w$	$-w$
N-Hash	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$-f$	$-w$
RIPEMD	$+w$	$+w$	$-w$	$+w$	$-f$	$+w$	$+w$	$-w$
SHA-1	$+w$	$+w$	$-w$	$+w$	$-f$	$+w$	$+w$	$-w$
Snefru	$+w$	$+w$	$-w$	$-w$	$-f$	$+w$	$-f$	$-w$
Tiger	$+w$	$+w$	$-w$	$+w$	$+w$	$+w$	$+w$	$-w$
Whirl	$+w$	$+w$	$-w$	$+w$	$+w$	$+w$	$-f$	$-w$
DES	$-w$	$-w$	$-w$	$-f$	$-w$	$-w$	$-w$	$-w$

Abbildung 7.5 Heuristiktest mit kryptographischen Hashfunktionen

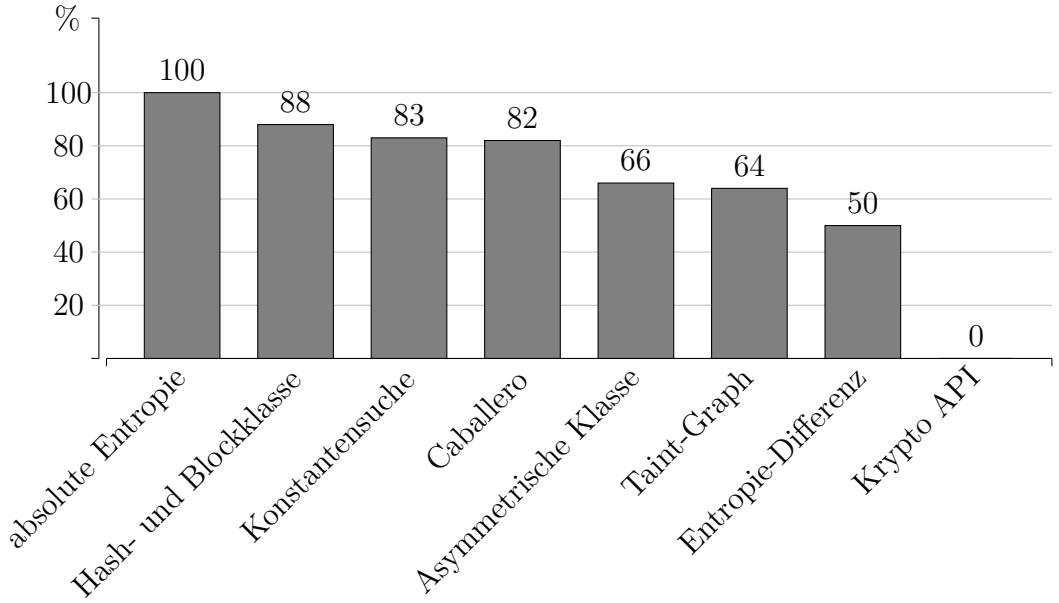

Abbildung 7.6 Richtige Positive Tests, bezogen auf die *jeweils* zu erkennenden Algorithmen

7.3.5 Vorauswertung

Die Phase 1 der Evaluation zeigt, dass die Erkennung asymmetrischer Kryptoalgorithmen verhältnismäßig schwieriger ist, da die gemessenen Eigenschaften durch Instruktionssequenzen erfüllt werden, die nicht ausschließlich von der zu erkennenden Algorithmenklasse verwendet werden. Kandidaten für eine mögliche approximative Lösung des Allgemeinen Erkennungsproblems sind die absolute Entropiemessung und die Caballero-Heuristik, da diese konzeptionell nicht auf bestimmte Algorithmenklassen ausgerichtet sind und gute Erkennungsleistungen aufweisen. Verwunderlich ist, dass die Heuristik zur Erkennung von Hashalgorithmen und symmetrischen Kryptofunktionen insgesamt mehr Algorithmen erkannt hat als die Caballero-Heuristik, da fälschlicherweise asymmetrische Kryptoroutinen angezeigt wurden. Diese Anomalie wurde vom Autor daraufhin überprüft. Es stellte sich heraus, dass der Schwellenwert von 40 Prozent dort mehrmals bei Elgamal und DSA überschritten wurde. Daraufhin fand ein Vergleich der gemessenen Anteile bei den Routinen der eigentlich zu erkennenden Klasse statt. Dieser zeigte, dass der Schwellenwert auf 50-60 Prozent hätte hochgesetzt werden können, was die falschen Positiven bei asymmetrischen Algorithmen hätte verhindern können. Da eine Sensitivitätsanalyse im Rahmen der Arbeit jedoch nicht möglich war, kann diese Erkenntnis zumindest in Zukunft genutzt werden. Ein Vergleich der Erkennungsmethode von Hashalgorithmen und symmetrischen Kryptofunktionen mit der Heuristik zur Erkennung asymmetrischer Algorithmen zeigt Ansätze einer Möglichkeit zur Einteilung erkannter Routinen durch Lösungen des klassenspezifischen Erkennungsproblems. Unerwartet niedrig ist die Erfolgsrate der Entropiedifferenzmessung. Dies könnte darin begründet sein, dass es nicht möglich war, repräsentative Testdaten bei der Ausführung der Algorithmen zu gewährleisten. In Phase 2 und 3 wird dieser Frage weiterhin nachgegangen. Die Graphheuristik besitzt eine mittelmäßige Erkennungsleistung, weist aber darauf hin, dass die Idee der sich beeinflussenden Speicherblöcke weitergehend geprüft werden sollte. Zur Nutzung und Verbesserung der Konstantensuche

haben die ersten Evaluationergebnisse bereits einige Erkenntnisse liefern können. Es ist wichtig, die einzelnen Muster möglichst kurz zu halten, um Algorithmen sicher finden zu können. Darüber hinaus könnte die Einführung von Schwellenwerten den daraus resultierenden höheren falschen Positiv-Raten besser entgegnen, da bei einer korrekten Erkennung während der Testdurchläufe im Gegensatz zu den falschen Positiven immer mehrere Suchmuster gefunden wurden. Die anfängliche Aussage, dass Softwareentwickler aus portabilitätsgründen eher Implementationen kryptographischer Algorithmen in Bibliotheken nutzen, als Aufrufe in die Windows Krypto API zu tätigen, kann bisher nicht gestützt werden, da Referenzimplementationen offensichtlich die eigenen Algorithmen nutzen. Die Ergebnisse der Methode zur Erkennung asymmetrischer Kryptosysteme zeigt bisher keine falschen Positive, erkannte jedoch nicht alle Algorithmen der zu erkennenden Klasse. Bezüglich der asymmetrischen Kryptosystme konnte Phase 1 zukünftigen Bedarf von Tests zur Erkennung dieser Algorithmenklasse mit entropiebasierten Methoden aufzeigen. Eine Übersicht der Erkennungsraten wird in Abbildung 7.6 illustriert. Bei der Interpretation ist es wichtig zu beachten, dass die Erkennungsraten nur anhand der Anzahl *jeweils zu erkennender Algorithmen* berechnet wurden, da ansonsten Methoden benachteiligt würden.

7.4 Evaluation mit bekannter Software

Um die Heuristiken mit einer möglichst großen Anzahl von Algorithmen zu testen, wurden in der ersten Evaluationsphase selbst erstellte Programme genutzt. Bei der Auswertung der Erkennungsmethode mittels Entropiedifferenz wurde jedoch vermutet, dass die Testdaten nicht repräsentativ genug seien. Weiterhin ist die Frage nach der Raten falscher Positive noch nicht ausreichend beantwortet. In Phase 2 werden daher fertige Programme getestet, welche keine kryptographischen Algorithmen verwenden. Daraufhin dient eine Menge von Programmen, die kryptographische Algorithmen nutzt, zum Testen der Anzahl richtiger Positive. Es wird erwartet, dass die praxisnäheren Tests mit fertigen Programmen zum einen repräsentativere Daten verarbeiten und zum anderen die praktische Anwendbarkeit der Erkennungsmethoden und des Evaluationssystems aufzeigen, da die Programme nicht speziell für die Evaluation angepasst wurden.

7.4.1 Erkennungsleistung

Zur Prüfung der Erkennungsleistung in praktischen Szenarien wurden vier Programme ausgewählt, die kryptographische Algorithmen zur Verschlüsselung von Dateien oder Netzwerkverkehr nutzen. Bei keinem der Testfälle waren alle angewandten kryptographischen Algorithmen dem Autor im Vorfeld bekannt. Tabelle 7.7 zeigt die Auswertung der Analysen. Bei allen Programmen konnten, soweit vom Autor nachprüfbar, alle genutzten kryptographischen Algorithmen ohne genaueres Wissen über die jeweilige Applikation gefunden werden. Wie in vorherigen Evaluationsphasen hat die absolute Entropiemessung alle kryptographischen Codeteile erfolgreich erkannt. Die Rate falscher Positive liegt zwar weit über dem Durchschnitt, dafür wurden die RSA- und die RC4-Implementationen jedoch ausschließlich von dieser

Heuristik vollständig erkannt. Im Vergleich dazu liegt die Erkennungsmethode mittels Entropiedifferenzmessung deutlich dahinter. Die Ergebnisse decken sich in etwa mit den Erkennungsraten aus Phase 1. Die Caballero-Heuristik besitzt ein gutes Verhältnis zwischen richtigen und falschen Positiven und ist daher im Ergebnis besser als die instruktionsbasierte Erkennung von Blockchiffren und kryptographischen Checksummen, die mit Ausnahme des RSA-Algorithmus alle anderen Codestellen hätte erkennen sollen. Die Methode zur Erkennung von asymmetrischen Kryptoalgorithmen hat RSA nicht erkannt und ausschließlich falsche Positive erzeugt. Da jedoch nur ein einziger asymmetrischer Algorithmus vertreten war, sollten daraus keine weiteren Schlüsse gezogen werden. Es zeigt sich, dass die Grapherkennung mit der höchsten Rate falscher Positive und einer verhältnismäßig niedrigen Rate richtiger Positive als schlecht zu bewerten ist. Keine der Programme hat die Windows-API zur direkten Ver- oder Entschlüsselung genutzt, sodass keine relevanten Funktionsaufrufe durch das Hooking registriert wurden. Die Konstantensuche hat alle zu erkennenden Algorithmen erfolgreich angezeigt, ohne dabei Falschmeldungen zu generieren. Das Verhältnis zwischen richtigen und falschen Positiven ist in Abbildung 7.9 in Anlehnung an die soeben beschriebenen Ergebnisse nochmal zusammenfassend dargestellt.

Programm	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking	Kryptofunktionen
Aphex Crypter	1/0	0/0	0/0	0/0	1/1	2/2	0/3	0/0	2(RC4)
File Encrypter	3/0	3/1	0/5	3/0	1/1	3/15	2/42	0/0	2(AES), 1(SHA1)
Curl HTTPS	2/0	2/1	0/2	2/0	3/12	3/42	2/92	0/0	1(3-DES), 1(SHA1), 1(RSA)
Aescrypt	4/1	4/1	0/8	4/0	0/2	4/3	1/3	0/0	2(AES), 2(SHA256)
AES File Crypter	1/2	1/2	0/1	1/1	1/3	1/3	1/3	0/0	1(AES)
Gesamt	11/3	10/5	0/16	10/1	6/19	13/65	6/143	0/0	13

Abbildung 7.7 Heuristiktest im Format (richtige Positive/falsche Positive)

7.4.2 Evaluation falscher Positive

Bei der vorherigen Auswertung der Erkennungsleistungen unter praxisnahen Bedingungen wurden bereits Raten falscher Positive in betracht gezogen. Da diese aber eventuell durch den in der jeweiligen Anwendung befindlichen kryptographischen Code hätten beeinflusst werden können, wurde eine separate Analyse von Raten falscher Positive mit Programmen ohne kryptographische Algorithmen durchgeführt.

Die Auswahl der Programme sollte hauptsächlich drei Verhaltensweisen abdecken, die in einem Großteil normaler Anwendungssoftware zu Finden sind:

- Netzwerkkommunikation
- Nutzung der Windows API
- Anwendung von Komprimierungsverfahren

Tabelle 7.8 zeigt die Ergebnisse der Analysen. Auffällig ist die verhältnismäßig hohe Rate falscher Positive bei der Anwendung von Komprimierungsverfahren während der Analyse des Programms "uharc". Die erheblich geringeren Werte bei dem Dateikomprimierungsprogramm "7-zip" sind dadurch zu erklären, dass sich die Kompressionsimplementation nur über einige wenige Funktionen erstreckt und der verwendete Algorithmus aufgrund der Anwendung von Baumstrukturen weniger arithmetische und bitbasierte Operationen enthält als das für Multimedia Inhalte entwickelte Programm "uharc". Einfache Windows-Programme, die fast ausschließlich aus API-Aufrufen bestehen und Daten im eigentlichen Hauptprogramm kaum modifizieren, besitzen die geringste Rate falscher Positive.

Die kleinere Meldungsanzahl der Erkennung von Blockchiffren und Hashalgorithmen mithilfe von Instruktionsanalysen im Vergleich zur Caballero-Heuristik ist dadurch zu erklären, dass erstere eine wesentlich kleinere Menge von Befehlen misst. Die Caballero-Heuristik betrachtet hingegen viele Instruktionen, wie beispielsweise "cmp" zum Vergleich von Registerwerten, die bei Kompressionsalgorithmen zu höheren Falscherkennungsraten führen. Die 85 fälschlicherweise gemeldeten Funktionen der Graphheuristik sind für eine in der Motivation vorgesehene manuelle Prüfung deutlich zu hoch. Gleiches gilt für die Methode der absoluten Entropiemessung. Die restlichen Heuristiken besitzen eine für praktische Belange akzeptable Anzahl falscher Positive. Die Frage ob und wie die Rate noch verbessert werden kann, soll in der Evaluationszusammenfassung aufgegriffen werden.

7.4.3 Vorauswertung

Die praktischen Tests der Heuristik mithilfe von Programmen, deren genutzte kryptographische Algorithmen dem Autor selbst nicht vollständig bekannt waren, zeigten zum einen, dass eine Erkennung relevanter Codeteile im Rahmen der Verifizierbarkeit durch den Autor funktioniert. Sie zeigen aber auch, dass einige Heuristiken zu hohe Raten falscher Positive besitzen, um die angezeigten Ergebnisse manuell effizient verifizieren zu können. Bei der Graphmethode hat dies wenig Einfluss auf das Gesamtergebnis, da die Erkennungsmethode von kryptographischen Hashes und Blockchiffren bisher gute Ergebnisse zeigt und beide Methoden die gleiche Menge an Algorithmen erkennen sollen. Die Anzahl falscher Positive der absoluten Entropiemessung ist ebenfalls hoch, dafür zeigen die vorigen Auswertungen jedoch, dass man ohne die Methode der absoluten Entropiemessung einige Codeteile nicht hätte finden können. Eine Möglichkeit wären zukünftige Tests mit höheren Entropie-Schwellenwerten. Diese konnten aufgrund der zeitlichen Beschränkung im Rahmen dieser Arbeit nicht mehr wiederholt werden. Angenommen, diese Evaluation würde zeigen, dass die Erkennungsrate bei einer Anhebung dieser Schwellenwerte deutlich

Algorithmus	Caballero	Hash/Block Klasse	Asymmetrische Klasse	Konstantensuche	Entropie Differenz	Absolute Entropie	Taint-Graph Block	Krypto API Hooking	Anzahl Funktionen
curl	0	1	2	0	3	10	42	0	532
notepad	0	0	0	0	2	2	1	0	34
calc	0	1	0	0	2	5	6	0	91
wget	1	3	5	0	3	12	13	0	347
nslookup	1	1	0	0	3	5	0	0	49
7-zip	0	0	1	0	2	9	7	0	335
uharc	27	10	17	3	9	17	12	0	301
telnet	0	1	0	0	3	9	4	0	64
Gesamt	29	17	25	3	27	69	85	0	1753
Prozentual	1.7	1.0	1.4	0.17	1.5	3.9	4.9	0	100

Abbildung 7.8 Tests auch falsche Positive mit bekannten Programmen

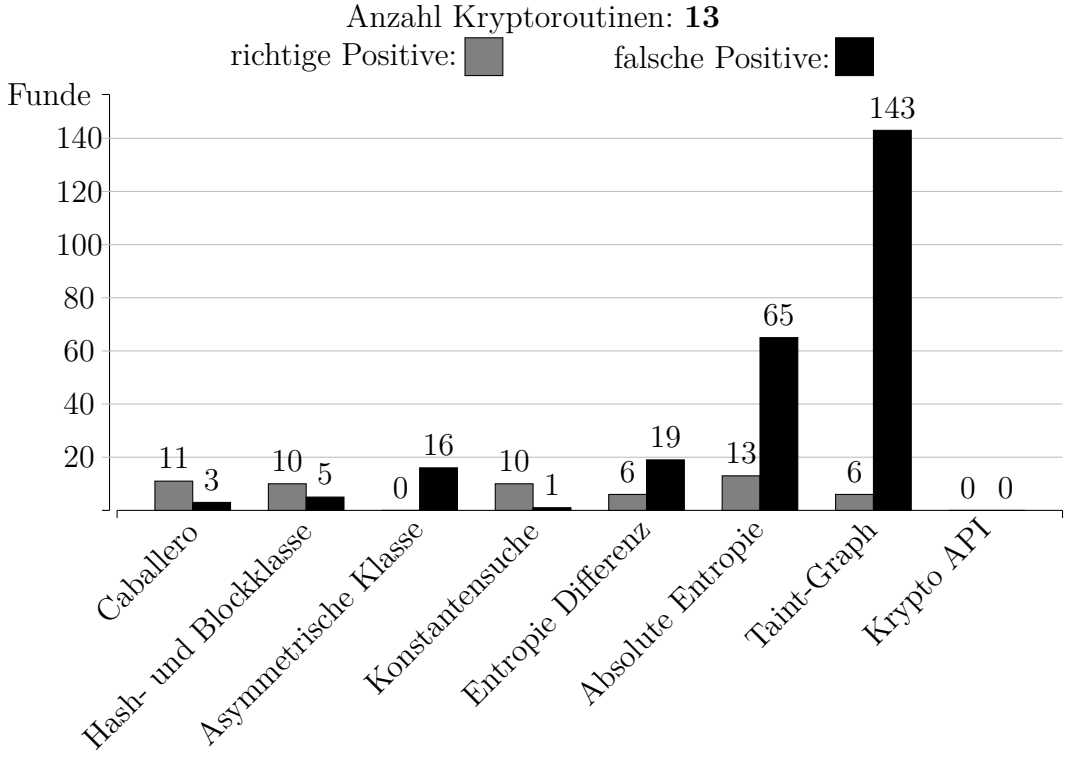

Abbildung 7.9 Verifikationsaufwand der Heuristikausgaben (siehe Tabelle 7.7)

sinkt. Dann wäre eine die absolute Entropiemessung trotzdem nützlich bei der Anwendung der unter Abbildung 4.1 beschriebenen Konzepte zur Kombination von Heuristikausgaben in Form von Formeln der Prädikatenlogik zur Reduktion von falschen Positiven. In der Gesamtauswertung der Evaluation wird auf dieses Verfah-

ren daher nochmal eingegangen. Alle weiteren Erkennungsmethoden zeigten keine nennenswerten Anomalien.

7.5 Fallstudie mit HLUX

Zur Prüfung der praktischen Anwendbarkeit der Heuristiken in Bezug auf die Einleitungsmotivation, wurde die Evaluationsplattform mit den Erkennungsmethoden genutzt, um Schadsoftware zu analysieren und darin Instruktionssequenzen kryptographischer Algorithmen zu finden.

HLUX ist ein Botnetz, das sich ausschließlich über den Versand von E-Mails verbreitet[22]. Über ein Peer-to-Peer Netzwerk sind die infizierten Computer organisiert und können von den unauthorisierten Betreibern zur Teilnahme an Spam Kampagnen oder Distributed Denial of Service Attacken (DDOS)[6] genutzt werden. Im Rahmen der Fallstudie wurde eine Instanz der Software in der virtuellen Maschine des Evaluationssystems über mehrere Stunden ausgeführt. Im Anschluss wurde das Programm IDA Pro[9] zur Prüfung der Ausgaben mithilfe einer statischen Analyse genutzt. Da die ausführbare Datei gepackt war, half die Möglichkeit zur Erzeugung eines Speicherabbildes . Dieses konnte in IDA[9] rekonstruiert werden, sodass die Heuristikausgaben mit vertretbarem Aufwand verifizierbar waren.

Bestätigt wurde die Existenz des SHA-1 Algorithmus in HLUX durch die Caballero-Heuristik, die Erkennungsmethode für Blockchiffren und Hashalgorithmen, der Konstantensuche sowie der absolute Entropieanalyse. Weitere Algorithmen konnten nicht gefunden werden. Dies war verwunderlich, weil laut Aussagen der Antivirenindustrie asymmetrische Kryptoalgorithmen zur Signaturprüfung von Updates genutzt werden. Weiterhin soll eine symmetrische Blockchiffre zur Verschlüsselung der Kommunikation innerhalb des Peer-to-Peer-Netzwerkes zum Einsatz kommen. Eine Prüfung des erzeugten Netzwerkverkehrs ergab, dass während der Ausführung keine Kommunikation mit dem Botnetz zu stande kommen konnte. Dies könnte erklären, warum keine symmetrischen Kryptoalgorithmen gefunden wurden, zeigt jedoch auch die praktisch relevanten Limitierungen der dynamischen Analyse.

Der Test zeigt, dass sowohl die Evaluationssoftware als auch die Heuristiken geeignet sind, um kryptographische Routinen in Software zu finden. Die Limitierung der dynamischen Analyse ist kein Argument für eine statische Analyse, da diese aufgrund der Tatsache, dass HLUX gepackt war, ohne vorherigen Aufwand zum Entpacken überhaupt nicht den eigentlichen Code hätte analysieren können.

7.6 Gesamtauswertung

Das Ziel dieser Evaluation ist es, über Aussagen einzelner Heuristiken hinaus auch Lösungsmöglichkeiten von Kombinationen mehrerer Heuristiken abzuleiten und somit die Erkennungsleistung insgesamt zu verbessern. Phase 1 der Evaluation hat die Erkenntnis gebracht, ob die Designziele der jeweiligen Heuristik bestätigt werden können. Im folgenden werden die Erkenntnisse zusammengefasst:

1. Die Caballero-Heuristik erkannte 82 Prozent aller kryptographischen Algorithmen und besitzt eine verhältnismäßig geringe Rate falscher Positive.

2. Die Heuristik zum Erkennen von kryptographischen Hashfunktionen und symmetrischen Kryptoalgorithmen erkennt 88 Prozent dieser Klasse, zeigte jedoch falsche Positive bei der Ausführung asymmetrischer Kryptoalgorithmen.

3. Die Heuristik zur Erkennung asymmetrischer Kryptosysteme erkannte $\frac{2}{3}$ dieser Algorithmen korrekt und zeigte keine falschen Positiven bei anderen kryptographischen Algorithmen.

4. Die Konstantensuche konnte kryptographische Algorithmen mit 83 Prozent richtigen Positiven identifizieren. Die falschen Negativen sind dabei zum Großteil auf eine nachteilige Wahl der Muster zurückzuführen, die in zukünftigen Implementierungen auf Basis der Evaluation verbessert werden kann.

5. Die Erkennung per Entropiedifferenzmessung erkannte mit insgesamt 50 Prozent verhältnismäßig wenige kryptographische Algorithmen.

6. Die Erkennung per absoluter Entropiemessung erkannte alle kryptographischen Algorithmen.

7. Die Taint-Graph-Heuristik erkannte mit 64 Prozent deutlich weniger kryptographische Algorithmen als die Heuristik zum Erkennen von kryptographischen Hashfunktionen und symmetrischen Kryptoalgorithmen. Beide Heuristiken sollten die gleiche Menge erkennen.

8. Die API-Heuristik konnte keine Aufrufe erkennen, da diese von den Testprogrammen nicht durchgeführt wurden.

In Phase 2 der Evaluation stand zum einen die Bewertung der Heuristiken anhand der jeweiligen Raten falscher Positive im Mittelpunkt. Zum anderen sollte die praktische Nutzbarkeit der Methoden durch die Ausführung von Programmen getestet werden, dessen genaue Funktionsweise dem Autor nicht bekannt war. Hierzu sind die absoluten und prozentualen falschen Positive in Abbildung 7.8 zusammenfassend dargestellt. Zusätzlich half Abbildung 7.9 beim Vergleich der Anzahl richtiger Positive zu falschen Positiven. Dies ist wichtig, da die relevanten Codestellen im Nachhinein manuell geprüft werden müssen. Hierbei hat sich gezeigt, dass zwei Heuristiken nicht zur direkten Erkennung genutzt werden sollten:

1. Die Graph-Heuristik erzeugt zu viele falsche Positive und kann durch die Erkennungsmethode der symmetrischen Kryptoalgorithmen und Hashfunktionen mittels Instruktionsanalyse ersetzt werden.

2. Die absolute Entropiemessung erzeugt zu viele falsche Positive, kann aber bei Kombinationen von Heuristiken zur Reduktion falscher Positive genutzt werden, da sie keine falschen Negativen aufwies.

Akzeptable Raten falscher Positive in praktischen Anwendungsfällen hängen vor allem von den zu akzeptierenden Kosten einer Analyse und der Wichtigkeit ab, alle Routinen zu finden. Phase 2 zeigte jedoch, dass die Anwendung der zur Erkennung

kryptographischer Routinen ausgeschlossenen Methoden einen Großteil des Verifikationsaufwands erzeugten und den Evaluationsprozess somit deutlich verlangsamten. In Fällen, in denen die Rate falscher Positive der Methode mittels absoluter Entropiemessung akzeptabel ist, sollte sie auf jeden Fall angewendet werden, da sie nach Messungen der Phase 1 mit 100 Prozent die beste Rate richtiger Positive besitzt. Im folgenden wird aber davon ausgegangen, dass diese Heuristik aufgrund der Kosten falscher Positive nicht zum Anzeigen kryptographischer Routinen eingesetzt werden kann.

In Kapitel 4 ist eine Methode illustriert, die es erlaubt, mehrere Heuristiken zu kombinieren, um Erkennungsleistungen gegebenenfalls zu verbessern. Aus den Ergebnissen der Evaluation wurden - auf Grundlage von Definition 5 - folgende Kombinationen abgeleitet:

Approximative Lösungen der Erkennungsprobleme
Sei Pk_1 die Menge asymmetrischer Kryptosysteme. Sei Pk_2 die Menge der symmetrischen Kryptoalgorithmen und der kryptographischen Hashfunktionen.
Für alle x im folgenden gelte $x \in S$ (siehe Definition 5).
P_g beschreibt das allgmeine Erkennungsproblem kryptographischer Algorithmen.
$P_{i,j}$ beschreibt ein algorithmenspezifisches Erkennungsproblem.
Sei H_a die Menge der von der absoluten Entropiemessung angezeigten Instruktionssequenzen.
Sei H_d die Menge der von der Entropiedifferenzmessung angezeigten Instruktionssequenzen.
Sei C die Menge der durch die Caballero-Heuristik angezeigten Instruktionssequenzen.
Sei $K_{i,j}$ die Menge der Instruktionssequenzen, des Algorithmus j aus der Klasse i.
Sei B die Menge der von der Heuristik zur Erkennung von symmetrischen Kryptoalgorithmen und Hashfunktionen angezeigten Instruktionssequenzen.
Sei A die Menge der von der Heuristik zum Finden asymmetrischer Kryptosysteme angezeigten Instruktionssequenzen.

Aus den Ergebnissen der Evaluation lassen sich folgende Formeln zur Kombination von Erkennungsmethoden ableiten:

1. Allgemeine Reduktion von falschen Positiven:
   $\forall x(\neg H_a x \rightarrow \neg P_g x)$
   Da H_a laut Evaluationsergebnissen alle relevanten Codeteile beinhaltet.

2. Lösung des allgemeinen Erkennungsproblems:
   $\forall x((Cx \wedge H_a x) \rightarrow P_g x)$
   Da C laut den Evaluationsergebnissen viele relevante Codeteile beinhaltet und gleichzeitig eine niedrige Anzahl falscher Positive aufwies, die durch den Schnitt mit H_a höchstens kleiner werden kann.

3. Lösung des algorithmenspezifischen Erkennungsproblems:
   $\forall x((K_{i,j} x \wedge H_a x) \rightarrow P_{i,j} x)$
   $K_{i,j}$ zeigte gute Erkennungsraten. Falsche Positive können durch den Schnitt mit H_a reduziert werden.

4. Erkennung der Menge asymmetrischer Kryptosysteme:
   $\forall x((Ax \wedge H_d x \wedge H_a x) \rightarrow Pk_1 x)$
   H_d und H_a zeigten gute Erkennungsraten. Zur Abgrenzung von anderen Algorithmenklassen muss A hinzugezogen werden.

5. Erkennung der Menge symmetrischer Chiffren und kryptographischer Hashfunktionen:
   $\forall x((Cx \wedge Bx \wedge H_a x \wedge \neg Ax) \rightarrow Pk_2 x)$
   A wird zur Abgrenzung zu asymmetrischen Kryptosystemen benutzt, da keine symmetrischen Kryptoalgorithmen oder Hashfunkionen angezeigt wurden. B zeigte gute Ergebnisse bei der Erkennung dieser Klasse. Da C und B gleich viele Codeteile dieser Klasse erkannt haben, kann C zusätzlich zur Reduktion von falschen Positiven eingesetzt werden.

Die Ergebnisse können in zukünftigen Arbeiten herangezogen werden, um mit den existierenden Heuristiken eine möglichst gute Lösung der definierten Erkennungsprobleme zu erreichen.

8

Zusammenfassung

Diese Arbeit zum Thema "Evaluation von Heuristiken zur Erkennung von Kryptographie in Software" basiert auf den Konzepten vorangegangener Arbeiten und dient der Zusammenfassung, Weiterentwicklung und Evaluation von Erkennungsmethoden. Die Problemstellung wurde weitergehend formalisiert und auf mehrere Problemklassen aufgeteilt, um die Erkennungsmethoden besser vergleichen und im Endergebnis sinnvolle Kombinationen von Heuristiken ableiten zu können. Auf Basis eines selbst entwickelten Evaluationsframeworks, das einige wichtige Vorteile gegenüber bisherigen Implementationen besitzt, wurden die Heuristiken entwickelt, implementiert und getestet. Der Evaluation vorausgegangene Tests haben gezeigt, dass die Heuristikparameter und Schwellenwerte anderer Autoren in den Anwendungsfällen der hier behandelten Thematik, wie beispielsweise durch Abbildung 5.6 gezeigt wurde, neu überdacht werden mussten. Hierdurch sind Fehler vorangegangener Arbeiten korrigiert worden. Die Ergebnisse bestätigten dieses Vorgehen. Die sowohl praxisnah, als auch repräsentativ gestaltete Evaluation hat ergeben, dass Implementationen kryptographischer Algorithmen in Software auch ohne das Vorhandensein von Quellcode durch Teilautomatisierung effizienter gefunden werden können als bei Anwendung manueller Reverse-Engineering-Techniken. Hierdurch hebt sich die Arbeit von vorangegangenen Arbeiten ab, da dort entweder nicht repräsentativ oder nicht praxisnah genug evaluiert wurde. Welche Erkennungsmethoden in welcher Kombination eingesetzt werden können, muss je nach Anwendungsfall entschieden werden. Die Arbeit kann an dieser Stelle Entscheidungsgrundlagen liefern. Die Konzepte des Evaluationssystems können von anderen aufgegriffen werden, um performantere Werkzeuge für die Programmanalyse zu entwickeln. Die starke Fokussierung auf Portabilität fördert eine lange Nutzbarkeitsdauer, die bei den meisten heutigen Implementationen von Software zu Zwecken der Programmanalyse nicht gegeben ist.

Literaturverzeichnis

[1] Aho, A. V., and Corasick, M. J. Efficient string matching: an aid to bibliographic search. *Commun. ACM 18* (June 1975), 333–340. http://doi.acm.org/10.1145/360825.360855.

[2] Aoki, K., Franke, J., Kleinjung, T., Lenstra, A. K., and Osvik, D. A. Factorization of the 1039th mersenne number. Mailing List, 2007.

[3] Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401.

[4] Böhne, L. Pandora's bochs: Automatic unpacking of malware, Jan. 2008.

[5] Caballero, J., Poosankam, P., Kreibich, C., and Song, D. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 621–634. http://doi.acm.org/10.1145/1653662.1653737.

[6] Cert, U. Understanding denial-of-service attacks. http://www.us-cert.gov/cas/tips/ST04-015.html.

[7] Clause, J., Li, W., and Orso, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 196–206. http://doi.acm.org/10.1145/1273463.1273490.

[8] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62. http://doi.acm.org/10.1145/1455770.1455779.

[9] Eagle, C. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[10] Felix, G. Automatic identification of cryptographic primitives in software, 2010.

[11] Foundation, F. S. Gmp - arithmetic without limitations. gmplib.org.

[12] Group, N. W. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, May 2008. http://tools.ietf.org/html/rfc5280.

[13] Guo, F., Ferrie, P., and Chiueh, T.-C. A study of the packer problem and its solutions. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), RAID '08, Springer-Verlag, pp. 98–115. http://dx.doi.org/10.1007/978-3-540-87403-4_6.

[14] Habib, I. Virtualization with kvm. *Linux J. 2008* (February 2008). http://dl.acm.org/citation.cfm?id=1344209.1344217.

[15] Intel. Vanderpool, July 2011. http://www.intel.com/technology/virtualization/.

[16] internet community, T., July 2011. www.winehq.org.

[17] ISO/IEC, and ITU-T. Abstract syntax notation one, July 2011. http://www.itu.int/ITU-T/asn1/index.html.

[18] Joan, Daemen Vincent, R. Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[19] Klein, T. All your private keys are belong to us, 2005.

[20] Kruh, L. *How to use the German Enigma cipher machine.* Artech House, Inc., Norwood, MA, USA, 1989, pp. 304–309. http://dl.acm.org/citation.cfm?id=95073.95190.

[21] Laboratories, R. public-key cryptography standards, July 2011. http://www.rsa.com/rsalabs/node.asp?id=2124.

[22] Labs, K. Hlux article. http://blog.antivirus365.net/?p=543.

[23] Lawton, K. P. Bochs: A portable pc emulator for unix/x. *Linux J. 1996* (September 1996). http://dl.acm.org/citation.cfm?id=326350.326357.

[24] Leder, F., Martini, P., and Wichmann, A. Finding and extracting crypto routines from malware, 2009.

[25] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40* (June 2005), 190–200. http://doi.acm.org/10.1145/1064978.1065034.

[26] Lutz, N. Towards revealing attackers intent by automatically decrypting network traffic, 2008.

[27] Matenaar, F. Asn.1 key extraction snippet. http://packetstorm.codar.com.br/crypt/SSL/sslfindkey.tar.gz.

[28] Microsoft. Dtours, July 2011. http://research.microsoft.com/en-us/projects/detours/.

[29] Microsoft. Windows api documentation, July 2011. http://msdn.microsoft.com.

[30] Microsoft. Windows crypto api, Aug. 2011. http://msdn.microsoft.com/en-us/library/aa380255%28v=vs.85%29.aspx.

[31] Miller, F. P., Vandome, A. F., and McBrewster, J. *Cyclic Redundancy Check: Computation of CRC, Mathematics of CRC, Error detection and correction, Cyclic code, List of hash functions, Parity bit, Information ... Cksum, Adler- 32, Fletcher's checksum.* Alpha Press, 2009.

[32] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42* (June 2007), 89–100. http://doi.acm.org/10.1145/1273442.1250746.

[33] Oliboni, C. Openpuff - tool for steganography, Aug. 2011. http://embeddedsw.net/doc/OpenPuff_Help_EN.pdf.

[34] Rice, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society 74*, 2 (1953), 358–366. http://www.jstor.org/stable/1990888.

[35] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21* (February 1978), 120–126. http://doi.acm.org/10.1145/359340.359342.

[36] Rutkowska, J. Red pill, Aug. 2011. http://invisiblethings.org/papers/redpill.html.

[37] Schneier, B. Quellcode aus dem buch 'applied cryptography', Aug. 2011. http://www.schneier.com/book-applied-source.html.

[38] Shannon, C. E. Communication theory of secrecy systems. Bell System Technical Journal, vol.28-4, 1949.

[39] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., James, N., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 1–25. http://dx.doi.org/10.1007/978-3-540-89862-7_1.

[40] Tan, M. A minimal gdb stub for embedded remote debugging, 2002.

[41] Wagner, R. A., and Fischer, M. J. The string-to-string correction problem. *J. ACM 21* (January 1974), 168–173. http://doi.acm.org/10.1145/321796.321811.

[42] Wang, Z., Jiang, X., Cui, W., Wang, X., and Grace, M. Reformat: automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European conference on Research in computer security* (Berlin, Heidelberg, 2009), ESORICS'09, Springer-Verlag, pp. 200–215. http://dl.acm.org/citation.cfm?id=1813084.1813102.

[43] Webster, A. F., and Tavares, S. E. On the design of s-boxes. In *Advances in Cryptology* (London, UK, UK, 1986), CRYPTO '85, Springer-Verlag, pp. 523–534. http://dl.acm.org/citation.cfm?id=646751.704578.

[44] Zhu, D. Y., and Chin, E. Detection of vm-aware malware, Dec. 2007.