Realisierung und Implementierung eines Algorithmus zur Echtzeit-Mustererkennung in einem Ethernet-Datenstrom

Peter Danielis[*], Stephan Kubisch[*], Dirk Timmermann[*]

[*] Universität Rostock, Institut für Angewandte Mikroelektronik und Datentechnik, Richard-Wagner-Str. 31, D-18051 Rostock, peter.danielis@uni-rostock.de

1. Einleitung

Während der letzten Jahre hat sich das Internet durch sein extremes Wachstum zu einem Massenmedium entwickelt, wodurch sich eine radikale Veränderung der Zielgruppe ergeben hat. Da jeder die Möglichkeit hat, sich Zugang zum Internet zu verschaffen, müssen sich Internetdienstanbieter zunehmend mit Sicherheitsaspekten und dem Schutz der Nutzer auseinander setzen. Verschiedene Sicherheitsmechanismen müssen inzwischen einen umfassenden Schutz vor Malware bieten und Angriffe von Hackern abwehren als auch das Netz selber schützen.

Intrusion Detection Systeme (IDS) haben sich daher seit den 80er Jahren als Ergänzung zu klassischen Sicherheitsmechanismen und Firewalls entwickelt. IDS gehen davon aus, dass konventionelle Sicherheitsmaßnahmen von Angreifern überwunden werden können und versuchen dieses zu erkennen. Dabei werden Angriffsmuster detektiert, die zum Beispiel spezielle Bitsignaturen sein können [Cla03].

Im Zuge dieser Arbeit wurden zusammen mit dem Industriepartner Nokia-Siemens Networks der Aho-Corasick- und der Set-Horspool-Algorithmus zur gleichzeitigen Erkennung von mehreren Bedrohungsmustern in einem Ethernet-Datenstrom realisiert. Da bei hohen Bandbreiten mit Softwarerealisierungen oder hybriden Ansätzen das Einhalten der notwendigen Geschwindigkeit zur Untersuchung des gesamten Netzverkehrs nicht mehr möglich ist, wurden die Algorithmen auf einem Virtex4-FPGA realisiert. Bei einer Hardwarelösung arbeiten alle Komponenten so schnell, dass die notwendige Rechenleistung zur Untersuchung des gesamten Netzverkehrs auf Bedrohungen erreicht wird. Dies wird als Wire-Speed bezeichnet. Zudem ist eine Hardwarelösung auf einem FPGA flexibel.

Das Dokument gliedert sich wie folgt: Die Algorithmen werden in Kapitel 2 erklärt. Die Hardwarerealisierung ist in Kapitel 3 aufgeführt. In Kapitel 4

werden für beide Algorithmen Resultate des funktionalen Tests auf der Zielplattform dargestellt. Kapitel 5 fasst das Dokument inhaltlich zusammen.

2. Algorithmen zur Mustererkennung

Die Suche nach einem Muster in einem Text wird als String Matching oder Pattern Matching bezeichnet. Dabei soll ein Muster *x[1..m]* der Länge *m* in einem Text *y[1..n]* der Länge *n* gesucht werden, wobei $n \gg m$ ist [Aho90]. Im Zuge dieser Arbeit handelt es sich bei Mustern um Zeichenketten, die Angriffsmustern entsprechen. Gesucht wird ein Klartext-Muster innerhalb eines Ethernet-Frames, das mit einem bekannten Suchmuster verglichen wird, welches in einer Menge von bekannten Angriffsmustern gespeichert ist.
Der naive Ansatz wäre es, zuerst an allen Stellen im Text zwischen *1* und *n-m+1* zu prüfen, ob Übereinstimmungen mit dem Muster vorliegen. Dieser Ansatz besitzt eine Speicherkomplexität von *O(m)*, um das Muster zu speichern und die jeweiligen zu vergleichenden m Zeichen des Textes. Die Zeitkomplexität beträgt *O(mn)*. Entsprechend der Ausführungen in [Aho90] und [Gie04] werden nachfolgend der Aho-Corasick- und der Set-Horspool-Algorithmus beschrieben, die sich besser als der naive Ansatz für die Mustersuche eignen. Diese eignen sich zudem für die Realisierung in Hardware [Dan06].
Bei dem Aho-Corasick-Algorithmus wird in einem Präprozess ein deterministischer, endlicher Automat konstruiert, der durch ein 6-Tupel der Form *(Q, Σ, g, f, output, q0)* beschreibbar ist.

- *Q* ist eine endliche Menge von Zuständen.
- *Σ* stellt ein endliches Eingabealphabet dar.
- *g* ist die Übergangsfunktion, die jedem Paar aus Zustand und Eingabezeichen einen Folgezustand oder *fail* zuordnet. Mit *fail* ist dabei eine Nichtübereinstimmung gemeint.
- *f* bezeichnet die Fehlerfunktion, die für jeden Zustand einen Folgezustand oder *fail* definiert.
- *output* ordnet einem Zustand eine Ausgabe zu.
- *q0* bezeichnet den Startzustand.

Der Aho-Corasick-Automat wird auch Baum oder Trie genannt, der aus den Schlüsselwörtern aufgebaut wird. In diesem kann während des Suchvorgangs gleichzeitig nach sämtlichen Schlüsselwörtern im Eingabetext gesucht werden. Dabei wird anders als bei normalen deterministischen, endlichen Automaten eine Fehlerfunktion *f* genutzt, die zum Einsatz kommt, wenn die normale Übergangsfunktion kein Ergebnis liefert.
In Abb. 1 ist beispielhaft ein Aho-Corasick-Trie dargestellt, in dem die Schlüsselwörter *he*, *hers*, *his* und *she* gespeichert sind. Graue Kreise kennzeichnen Zustände, wobei jeder Zustand eine Zustandsnummer besitzt. Der Zustand 0 stellt den Startzustand dar. Fett umrandeten Zuständen sind *output*-Funktionen zuge-

ordnet. Zustandsübergänge sind durch Pfeile gekennzeichnet, wobei gestrichelte Linien Fehlerfunktionen darstellen.

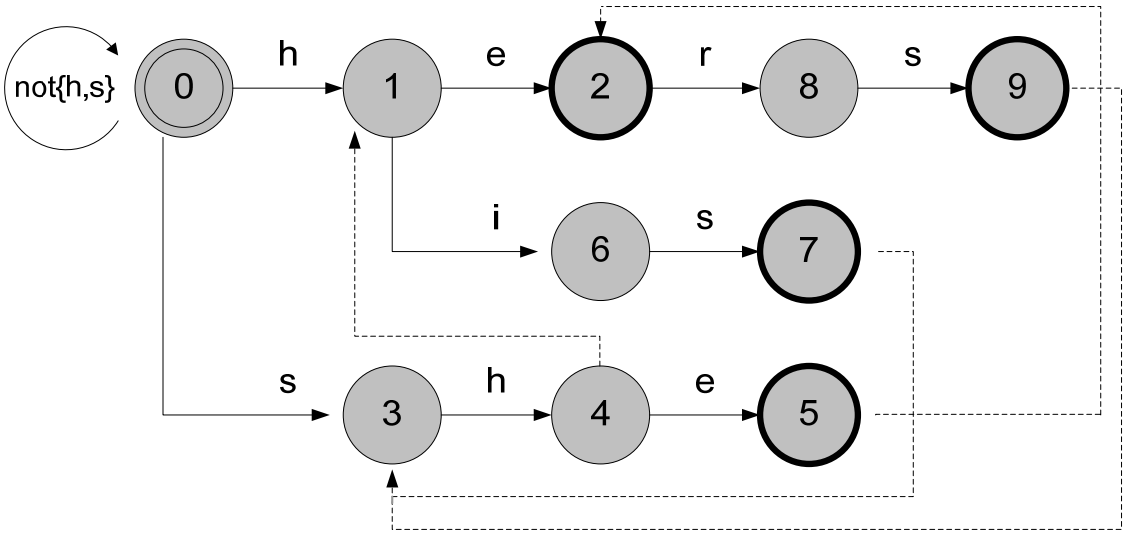

Abb. 1: Aho-Corasick-Trie mit den Schlüsselwörtern *he, hers, his* und *she*

Die Zeit- und Speicherkomplexität für die Erstellung des Tries beträgt $O(m)$. Bei der Suche handelt es sich um eine Vorwärtssuche im Trie, bei der jedes Zeichen im Eingabetext gelesen und verglichen wird. Die Zeitkomplexität für den Suchvorgang beträgt immer $O(n)$ und ist somit linear. Die Speicherkomplexität ergibt sich mit $O(m)$.

Der Set-Horspool-Algorithmus ist der Horspool-Algorithmus für die gleichzeitige Suche nach mehreren Mustern. Der Set-Horspool-Algorithmus baut einen Baum auf, in dem die Muster rückwärts eingetragen werden. Die minimale Länge aller Muster ℓ_{min} legt die Anfangsposition fest, an der im Eingabetext mit der Suche begonnen wird. Von dieser Position wird der Textabschnitt rückwärts gelesen und die Zeichen werden mit denen im Baum verglichen. Es handelt sich also beim Set-Horspool-Algorithmus im Gegensatz zum Aho-Corasick-Algorithmus um eine Rückwärtssuche, bei der nicht jedes Zeichen vom Text gelesen werden muss. Wenn Zeichen im Baum und im Eingabetext übereinstimmen, wird das nächste Zeichen in Baum und Text verglichen, bis ein Endzustand erreicht wird. Bei Nichtübereinstimmung wird für das erste verglichene Zeichen die Verschiebung über eine Tabelle ermittelt. Für Zeichen, die nicht im Baum vorkommen, steht in der Tabelle der Wert ℓ_{min}.

Analog zur Suche in einem Wort mit dem Boyer-Moore-Algorithmus kann die Tabelle für die Schiebefunktion in einem Präprozess mit einem Aufwand von $O(\sigma + r*\ell_{min})$ konstruiert werden [Chr04]. σ stellt die Anzahl der Zeichen in einem Alphabet dar und r ist die Anzahl der Schlüsselwörter. Die mittlere Anzahl von Vergleichen pro Suchphase beträgt $\log_\sigma r$ und somit ergibt sich die durchschnittliche Zeitkomplexität mit $O(\log_\sigma r)$. Besonders effizient ist der Set-Horspool-Algorithmus bei einer kleinen Anzahl von Mustern und einem relativ großen Eingabetext. Im schlechtesten Fall beträgt die Zeitkomplexität $O(n*\ell_{max})$,

wobei ℓ_{max} die maximale Länge der Schlüsselwörter darstellt. Die Speicherkomplexität beträgt *O(m)*.

3. Hardwarerealisierung

Die Realisierung des Designs erfolgte auf einem Xilinx ML405-Evaluation-Board. Es wurde Wert auf einen modularen Aufbau gelegt, so dass jede Komponente auf Grund einheitlicher Schnittstellen leicht austauschbar ist. Dies gilt insbesondere für die Komponente, die den jeweiligen Suchalgorithmus enthält.
Eingehende Datenpakete werden sowohl im Upstream als auch im Downstream in Synchronisations-FIFO-Komponenten zwischengespeichert. Eine Untersuchung von Datenpaketen erfolgt nur im Upstream. Aus der Synchronisations-FIFO-Komponente im Upstream werden die Daten byteweise ausgelesen und an eine Komponente zur Analyse des Datenpakets sowie einer Komponente, die einen der Algorithmen zur Mustererkennung enthält, weitergeleitet.
Die Komponente zur Analyse des Datenpakets teilt der Komponente mit dem Suchalgorithmus mit, ab welchem Byte es sich um zu vergleichende Bytes der Nutzlast handelt. Während des Vergleichs liest die Komponente mit dem Suchalgorithmus Vergleichsdaten aus einem Speicher und erhöht die Anzahl gefundener Muster, sobald eine Übereinstimmung festgestellt wurde. Ein LCD gibt Statusinformationen wie z. B. die Anzahl der gefundenen Muster aus. Zuvor wurden sämtliche Vergleichsdaten während eines Präprozesses von einer PowerPC-CPU in den Speicher geschrieben.

4. Testergebnisse

Für den Test des erstellten Designs wurden verschieden lange Ethernet-Frames mit der maximalen Übertragungsrate von 100 Mbit/s an das ML405-Board geschickt. Auf dem LCD wird dabei die Anzahl erkannter Muster angezeigt.
In die gesendeten Ethernet-Frames werden Angriffssignaturen eingebettet, die zuvor im Präprozess im Speicher abgelegt wurden. Bei den Ethernet-Frames handelt es sich um aufgezeichnete Datenpakete aus realem Datenverkehr, um ein möglichst realistisches Szenario zu gewährleisten. Im ersten und zweiten Testfall wurden unterschiedliche Frames mit einer Länge von 1314 Byte vom Typ TCP aufgezeichnet. Der dritte Testfall umfasst Frames des Typs UDP mit der Länge 132 Byte. Für jeden Testfall wurden 10.000 Frames mit jeweils einem Bedrohungsmuster hintereinander mit der maximalen Geschwindigkeit gesendet. Dieses wird jeweils für zwei, acht und 16 vorgespeicherte Bedrohungsmuster durchgeführt. Über die Anzeige auf dem LCD wird erfasst, wie viele Muster erkannt wurden.
In Abb. 2 ist die Abhängigkeit des Verhältnisses der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames von der Anzahl der im Präprozess vorgespeicherten Muster für Aho-Corasick und Set-Horspool graphisch dargestellt. Die Leistungsfähigkeit der Algorithmen ist dabei in erster Linie durch die Speicherzugriffszeit begrenzt. Speicherzugriffe werden notwendig, wenn ein Byte im

Ethernet-Datenstrom mit einem im Aho-Corasick- beziehungsweise Set-Horspool-Trie gespeicherten Knoten übereinstimmt. Beim Set-Horspool-Algorithmus fallen zudem zusätzliche Speicherzugriffe an, um die Verschiebungslänge für ein Zeichen aus dem Speicher zu lesen. Je mehr Speicherzugriffe vorgenommen werden müssen, desto geringer ist der Durchsatz der Ethernet-Frames. Die Ethernet-Datenpakete werden in den Synchronisations-FIFO-Komponenten zwischengespeichert und können verworfen werden, wenn nicht mehr genügend Platz zur Verfügung steht. Bei einem zu geringen Durchsatz kann es folglich zum Verwerfen von Frames kommen.

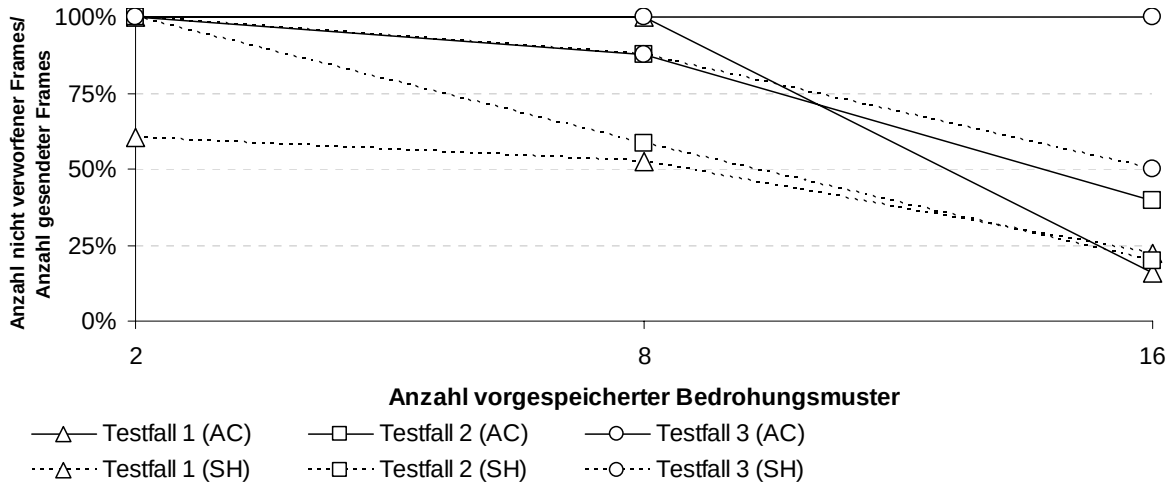

Abb. 2: Darstellung der Testergebnisse für Aho-Corasick und Set-Horspool

Mit steigender Anzahl vorgespeicherter Muster sinkt das Verhältnis der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames für beide Algorithmen. Dies war zu erwarten, da bei mehr im Präprozess generierten Bedrohungssignaturen auch die Wahrscheinlichkeit steigt, dass Übereinstimmungen mit Datenbytes der Ethernet-Frames auftreten und somit Speicherzugriffe notwendig werden. Dies hat wiederum das Verwerfen von ankommenden Datenpaketen zur Folge, wenn nicht mehr genügend Platz zur Pufferung in der empfangsseitigen Synchronisations-FIFO-Komponente zur Verfügung steht.
Für kurze Ethernetframes, die bei dem dritten Testfall gesendet wurden, ist das Verhältnis der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames höher als für lange Frames. Kurze Frames liegen früher komplett im Synchronisations-FIFO-Speicher vor und es kann somit früher mit dem Auslesen und dem Vergleich begonnen werden. Damit steht wieder neuer Speicherplatz für neue Frames zur Verfügung. Frames werden nur dann gespeichert und nicht verworfen, wenn bei deren Empfang noch genug Platz für die Speicherung des kompletten Datenpaketes vorhanden ist. Dies trifft bei kurzen Frames eher zu als bei langen Datenpaketen.
Im Vergleich mit dem Set-Horspool-Algorithmus stellt sich der Aho-Corasick-Algorithmus als leistungsfähiger heraus. Für eine geringe Anzahl vorgespeicherter Muster liegt der Aho-Corasick-Algorithmus für alle Testfälle bei einem Ver-

hältnis der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames von bis zu 100 %. Für den dritten Testfall beträgt das Verhältnis der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames bei Aho-Corasick stets 100 %.
Die Erhöhung der Anzahl von im Präprozess erzeugten Mustern führt mit Ausnahme des dritten Testfalls bei Aho-Corasick bei beiden Algorithmen zu einer Reduzierung des Verhältnisses der Anzahl nicht verworfener Frames zur Anzahl gesendeter Frames. Erklärung hierfür ist wiederum die größere Anzahl von notwendigen Speicherzugriffen, wodurch der Durchsatz geringer wird und die Anzahl verworfener Frames steigt.

5. Zusammenfassung und Ausblick

Als Ergebnis dieser Arbeit wurden zwei Algorithmen zur Mustererkennung in einem Ethernet-Datenstrom implementiert. Die Umsetzung erfolgte in VHDL auf dem Xilinx ML405-Evaluation-Board, das einen Virtex4-FPGA mit einer PowerPC-CPU sowie Peripheriegeräten und Speicher kombiniert. Bei der Analyse des Designs hat sich herausgestellt, dass die Leistungsfähigkeit der Algorithmen in erster Linie durch die Speicherzugriffszeit begrenzt ist. Je mehr Speicherzugriffe vorgenommen werden müssen, desto geringer ist der Durchsatz der Ethernet-Frames. Dadurch kann es zum Verwerfen von Frames kommen. Ein Vergleich der Ergebnisse des funktionalen Tests für den Aho-Corasick- und den Set-Horspool-Algorithmus zeigte dabei, dass der Aho-Corasick-Algorithmus diesbezüglich leistungsfähiger als der Set-Horspool-Algorithmus ist. In folgenden Arbeiten könnte das Design um eine adaptive Komponente erweitert werden, um gänzlich unbekannte Angriffe, deren Detektion bisher nicht möglich ist, mittels Anomalieerkennung zu detektieren.

6. Literatur

[Aho90] A.V. AHO. Algorithms for Finding Pattern in Strings, in Handbook of Theoretical Computer Science, Algorithms and Complexity, volume A. Elsevier Sience Publishers B.B., Amsterdam, 1990. Kapitel 5, ff. 255-300.

[Chr04] Christian Charras, Thierry Lecroq. Handbook of Exact String-Matching Algorithms. King’s College Publications, Februar 2004.

[Cla03] Thorsten Clausius. Intrusion Detection Systeme - Information Sources. Technische Universität Darmstadt, Seminararbeit, 11. Dezember 2003.

[Dan06] Peter Danielis. Realisierung und Implementierung eines Algorithmus zur Echtzeit-Mustererkennung in einem Ethernet-Datenstrom. Universität Rostock, Diplomarbeit, 10. Oktober 2006.

[Gie04] Nico Gierspeck. Multiple-String-Matching. Otto von Guericke Universität, Magdeburg, 14. Dezember 2004.