Ein neues Konzept zur Realisierung von Software-Kopierschutz

Klaus Kiefer

Bericht A/05/96

16. September 1996

Graduiertenkolleg Informatik, Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken (Germany)
e-mail: kiefer@cs.uni-sb.de

Betreuer: Prof. J. Buchmann (TH Darmstadt)

Inhaltsverzeichnis

1 Bisherige Strategien **1**

2 Neues Konzept **4**
2.1 Prinzipielle Strategie 4
2.2 Verfeinerung des Konzeptes 5
2.3 Ausblick 8

3 Spezifikation der Kryptokarte **9**
3.1 Kryptographische Anforderungen 9
3.2 Programmierung der Kryptokarte 12
3.3 Interner Aufbau 15
3.4 Schnittstellen 18

4 Erstellen geschützter Programme **22**
4.1 Aufbereitung von Programmcode 22
4.2 Erzeugen von Codevarianten 29

5 Schlußbemerkungen **30**

A Realisierungen **33**
A.1 Interpreter für Integerterme 33
A.2 Interpreter für boolsche Terme 34

1 Bisherige Strategien

Zur sicheren Realisierung von Software-Kopierschutz ist unserer Meinung nach eine zusätzliche Hardwarekomponente notwendig. Bei reinen Softwarelösungen, wie zB. einer Passwortabfrage oder dem Ausgeben einer Lizenzmeldung auf dem Bildschirm, reduziert sich das Problem, einen effektiven Kopierschutz zu realisieren, auf die Aufgabe, Codemanipulationen zu verhindern. Dazu muß auf den Code wie auf Daten zugegriffen werden, zB. um Vergleichsoperationen durchführen zu können. Da aber Code- und Datensegmente leicht zu trennen sind, können solche Zugriffe auf das Codesegment relativ einfach isoliert und gepatcht werden.

Ein detaillierter Überblick gebräuchlicher Kopierschutzsysteme findet sich in [6]. Die dort beschriebenen Methoden sind jedoch entweder unsicher oder benutzerunfreundlich. So sind auch Schutzmechanismen, die durch sogenannte Dongles geboten werden, nicht ausreichend. Hier werden meist nur irgendwelche Werte vom Dongle abgefragt, ohne daß dieser eine komplexe Funktionalität hat. Deshalb können die Zugriffe auf den Dongle normalerweise sehr einfach aus dem Programm entfernt werden [10]. Um einen effektiven Softwareschutz zu erreichen, sind kompliziertere Lösungen notwendig.

In den bisherigen Ansätzen wird vorgeschlagen, eine kryptographische CPU einzusetzen, die verschlüsselte Befehle einliest, danach diese intern entschlüsselt und abarbeitet, vgl. [2], [7] oder [8]. Eine solche CPU muß abhörsicher konzipiert sein, damit ein Angreifer nicht geheime Schlüsselinformationen auslesen kann. Abgesehen davon, daß dieses Konzept nur mit neu zu entwickelnden CPUs realisiert werden kann, halten wir mit Blick auf moderne Rechnerarchitekturen große Performanceeinbußen für unvermeidlich. Alle neueren CPUs wie zB. der Intel-Pentium nutzen das Prinzip des Instruktionspipelining. Eine notwendige Entschlüsselungseinheit würde durch eine oder (was realistischer ist) mehrere zusätzliche Stufen in der Pipeline realisiert werden. Allerdings ist der Codeverbrauch moderner CPUs weit höher als der Datendurchsatz verfügbarer Verschlüsselungshardware. Beim Ausführen von Befehlen, die in der Pipeline stehen, schafft die CPU eine Instruktion pro Takt. Bei einem 133 MHz Prozessor wären das also mindestens 133 MB/s - falls jede Instruktion nur ein Byte lang wäre, was sicherlich zu gering geschätzt ist. Der schnellste kommerzielle DES Chip schafft jedoch lediglich 35.5 MB/s (laut [12]; dabei sind die Durchsatzraten der Chips von VLSI Technology Inc. jedoch alle zu hoch angegeben - der VM007 zB. schafft

lediglich 24 MB/s [13]). Außerdem arbeiten die schnellen Kryptochips selbst ,,gepipelined". Das bedeutet aber, daß ein Neustart der CPU-Pipeline (nach einem Branch-Befehl) jetzt viel länger dauert, weil zuerst die Decryption-Pipeline komplett durchlaufen werden muß - es tritt ein großer Zeitverlust auf. Ferner ist bei Verwendung von Nichtregistervariablen der volle Overhead, der durch das Entschlüsseln entsteht, unvermeidlich. Die Folge: Datenzugriffe auf den Hauptspeicher werden extrem teuer.

Um Angreifern weitere Beobachtungsmöglichkeiten zu entziehen - auf welchen Maschinenbefehl wird gerade zugegriffen, gibt es Schleifen usw. - wurde eine Erweiterung des obigen Szenarios vorgeschlagen [5]. Dabei soll eine Oblivious Random Access Maschine simuliert werden, dh. die Speicherzugriffe der CPU sind zufällig und gleich verteilt für je zwei Programme mit selber Laufzeit. Jeder einzelne Speicherzugriff wird durch eine Folge von Speicheroperationen simuliert, welche die eigentliche Zieladresse verschleiern. Damit lernt ein Beobachter *nichts* bei der Ausführung des Programms außer dem Input/Output-Verhalten und der Rechendauer. Aber nicht nur, daß durch diese Strategie Lokalitätseigenschaften des Programms künstlich zerstört werden (die Lokalität von Programmen wird durch verschiedene Memorymanagement-Techniken wie zB. Caching zum Speedup der Programmausführung genutzt und wird daher besonders angestrebt), die Autoren geben selbst folgende Verzögerungsrate bei Anwendung ihres Verfahrens an: ein Programm, das normalerweise in t Instruktionen abgearbeitet wird, benötigt nun $O(t(\log_2 t)^3)$ Schritte. Die Ausführungszeit *vervielfacht* sich also um einen in t wachsenden Faktor. Wenn wir eine CPU mit 50 MIPS zugrundelegen und als Verzögerungsfaktor exakt $(\log_2 t)^3$ annehmen, so ergeben sich folgende Beispielwerte, die natürlich völlig abwegig sind:

normale Dauer	1 sec	1 min	10 min	1 h
oblivious RAM	4,5 h	3 Wochen	292 Tage	6 Jahre

Weiter sehen wir große organisatorische Schwierigkeiten bei Einsatz einer Krypto-CPU, zB. im Multitaskingbetrieb. Es sollten gleichzeitig verschlüsselte und unverschlüsselte Programme ablaufen können, dh. es müßte bei jedem Speicherzugriff entschieden werden, ob entschlüsselt wird oder nicht. Zur Abschwächung der oben beschriebenen Performanceprobleme könnte man ja die Krypto-CPU um einen - ebenfalls abhörsicheren - Cache-Speicher erweitern. Allerdings ist jedes Verschlüsselungsverfahren langsamer als reiner Datentransfer aus dem Hauptspeicher, dh. bei jedem ,,Cachefehler" müßte die CPU

relativ lange warten, da die Entschlüsselung nicht parallel zur Abarbeitung von Code stattfindet. Dies könnte man vielleicht noch durch parallelen Einsatz von Dechiffriereinheiten wettmachen, denn beim Caching werden jeweils größere Speicherblöcke (zB. Pages) ersetzt. Bislang funktionierte das Memorymanagement jedoch unabhängig von der Art der aktiven Prozesse, nun müßte unterschieden werden, ob bei Speicherzugriffen dechiffriert wird oder nicht, und - wenn ja - mit welchem Schlüssel (es sei denn, alle Programme eines Rechners seien einheitlich verschlüsselt, was allerdings Massendistribution von Code ausschlösse). Zudem sollte zur Vermeidung von Analogschlüssen ein probabilistisches Verschlüsselungsverfahren verwendet werden - dann sind aber Ciphertextblöcke länger als Klartextblöcke. Dies müßte vom Memorymanagement berücksichtigt werden. Zur Realisierung dieses Ansatzes genügt es also nicht, eine Krypto-CPU zu entwickeln, die sich einfach in bestehende Systeme einbinden ließe, sondern es müßten größere Teile von Rechnerarchitektur und Betriebssystem neu konzipiert werden. Abgesehen vom dafür notwendigen Entwicklungsaufwand wären die neuen Systeme komplizierter und damit teurer.

Außerdem sehen wir erhebliche Akzeptanzprobleme bei ehrlichen Nutzern, wenn überhaupt nicht mehr ersichtlich ist, was das erworbene Programm im eigenen Computer momentan gerade macht. Damit wären ,,trojanischen Pferden" Tür und Tor geöffnet, was insbesondere bei der zunehmenden Vernetzung von Rechnern zu großen Sicherheitsproblemen führen würde. Unserer Meinung nach wird ein effektiver Kopierschutz nicht erst erreicht, wenn ein Beobachter bei der Programmausführung *nichts* lernt. Unser Ziel ist: um ein geschütztes Programm zu ,,cracken", muß ein Softwarepirat den Programmablauf nachvollziehen und einigermaßen verstehen; automatisches Patchen soll unmöglich sein. Ist das zu schützende Programm komplex genug, so reicht das zum Kopierschutz. Diesen Schutz möchten wir erreichen mit Hilfe einer *optionalen* Erweiterungskomponente, mit der existierende Systeme aufgerüstet werden können. Diese Erweiterung wird konfigurationsunabhängig konzipiert sein und läßt somit dem Benutzer die Möglichkeit eines problemlosen Systemupgrades, zB. durch Einbau einer schnelleren CPU.

2 Neues Konzept

2.1 Prinzipielle Strategie

Wir schlagen also vor, eine zusätzliche Hardwarekomponente zu verwenden, die an bestehende Systeme angebunden werden kann, zB. als Steckkarte. Diese Komponente soll kryptographische Prozeduren wie Verschlüsselungen ausführen und dabei abhörsicher sein, dh. geheime Schlüsselinformationen dürfen nicht auslesbar sein. Weiter soll diese Zusatzhardware - die wir im folgenden kurz als **Kryptokarte** bezeichnen werden - einen Teil der Funktionalität des Programms übernehmen. Dazu wird eine einfache Assemblersprache definiert, mit welcher die Kryptokarte systemunabhängig programmiert wird. Das bedeutet, daß mit geringen Anpassungen (zB. dem Erstellen geeigneter Treiber) die Kryptokarte in unterschiedlichen Computersystemen eingesetzt werden kann. Außerdem kann die Kryptokarte von verschiedenen Programmen genutzt werden, und bei entsprechender Spezifikation auch von mehreren Programmen gleichzeitig. Da jede Kryptokarte individuell verschiedene Schlüssel benutzt, sind so geschützte Programme nur im Zusammenspiel mit einer einzigen Kryptokarte lauffähig. Damit wird die Weitergabe von unautorisierten Programmkopien sinnlos.

Die Funktionalität, die die Kryptokarte übernehmen soll, ist die Ablaufsteuerung der zu schützenden Programme. Dazu fassen wir ein Computerprogramm als Ansammlung von Unterroutinen auf. Dem gegenüber gibt es eine *Kontrollstruktur*, welche die Unterroutinen in einer bestimmten Reihenfolge aufruft, abhängig von einigen *Kontrollvariablen*. Die Unterroutinen sind in verschlüsselter Form im Hauptspeicher (oder ggf. auf der Festplatte) abgelegt, die Kontrollstruktur und die Kontrollvariablen befinden sich im Speicher der Kryptokarte, unsichtbar für den Angreifer. In der Karte wird also entschieden, welche Unterroutine als nächstes ausgeführt werden soll. Diese Unterroutine wird dann in die Kryptokarte geladen, entschlüsselt, zurück in den Hauptspeicher kopiert und dynamisch zum aktuellen Programm dazugelinkt - gleichzeitig werden abgearbeitete Unterroutinen aus dem Speicher entfernt. Damit realisieren wir ein *Overlay*-Programm, vgl. [1]. Die Kontrollvariablen werden von den Unterroutinen aus gesetzt bzw. verändert mittels verschlüsselter Nachrichten, die an die Kryptokarte geschickt werden.

Das zu schützende Programm wird also in relativ kleine Fragmente untergliedert, die als Unterroutinen aufgerufen werden. Der Wechsel zwischen

zwei Unterroutinen sollte an möglichst zeitunkritischen Stellen erfolgen. Beim Ablauf des Programms sieht der Angreifer immer nur kleine Abschnitte des Codes unverschlüsselt. Um das Programm patchen zu können, muß eine funktionierende Kontrollstruktur aufgebaut werden, die die beobachteten Unterroutinen in Abhängigkeit der richtigen Ereignisse aufruft. Der Angreifer muß den Programmablauf insoweit verstehen, um diese Abhängigkeiten zu erkennen. Diese Aufgabe sollte weiter erschwert werden, indem die Unterroutinen viele Nachrichten - auch nach irrelevanten Ereignissen - an die Kryptokarte schicken. Weiter werden die Messages probabilistisch verschlüsselt sein, um die Identifizierung gleicher Nachrichten zu erschweren.

Dieses Szenario ist unserer Meinung nach realisierbar ohne allzu großen Performanceverlust. Kryptokarte und CPU arbeiten parallel, dh. während die CPU ein Codefragment abarbeitet, bereitet die Kryptokarte die nächste Unterroutine vor. Liegt in der Kontrollstruktur eine Verzweigung vor, so kann die Kryptokarte „Branch-Preparation" betreiben, dh. alle Codevarianten, die auftreten könnten, werden vorbereitet. Wird von der Kryptokarte die nächste Unterroutine angefordert, so wird entsprechend dem Wert der Kontrollvariablen eine Codevariante ausgewählt und in den Hauptspeicher kopiert. Dies kann in vielen Systemen sehr effizient mittels Direct Memory Access (DMA) realisiert werden. In [9] wird beschrieben, wie die dynamische Modifizierung des Instruktionsspeichers effizient und sicher durchgeführt werden kann; dazu werden Schnittstellen-Prozeduren angegeben, die sich auf verschiedenen Plattformen (also Kombinationen von Hardware und Betriebssystem) als leicht portierbar erwiesen. Danach muß nur noch ein einfaches Linkverfahren durchgeführt werden, damit der neue Code zB. auf globale Variablen zugreifen kann. Dann kann in die Unterroutine ganz normal eingesprungen werden.

2.2 Verfeinerung des Konzeptes

Dem Angreifer soll es erschwert werden, durch Beobachten der Programmausführung bzw. der Aktionen der Kryptokarte Rückschlüsse auf die verborgene Kontrollstruktur ziehen zu können. Dies soll geschehen durch

- Polymorphie der Unterroutinen, dh. der jeweilige Code kann in verschiedenen Varianten auftreten.

- Oblivious Access auf verschlüsselte Unterroutinen, dh. die Speicherzugriffe der Kryptokarte werden verschleiert.

Um ersteres zu erreichen, werden in den Unterroutinen Codevarianten zusammen mit Metaregeln verschlüsselt abgespeichert. Wird die Unterroutine in der Kryptokarte entschlüsselt, so bestimmen die Metaregeln anhand weiterer Kontrollvariablen, welche Variante zusammengebaut wird. Diese zusätzlichen Kontrollvariablen werden keine Zufallsvariablen sein (sonst kann der Angreifer äquivalente Routinen erkennen, indem das Programm mehrfach mit gleichen Eingabewerten abläuft), sondern Variablen, die anhand ,,unwichtiger" Ereignisse gesetzt bzw. verändert werden (damit meinen wir Ereignisse, die keinen Einfluß auf Verzweigungen in der Kontrollstruktur haben). Je größer die Anzahl der möglichen Codevarianten ist, desto schwieriger wird es für den Angreifer, die Programmstruktur zu durchschauen. Dazu muß dieser zunächst erkennen, daß eine gerade abgearbeitete Routine funktional äquivalent zu einer früheren ist, und dann verstehen, nach welchen Kriterien die nächste Unterroutine ausgewählt wird.

Weiter sollen die Zugriffe der Kryptokarte auf den Hauptspeicher verschleiert werden. An jeder Stelle der Kontrollstruktur müssen eine oder (bei Verzweigungen) mehrere verschlüsselte Unterroutinen in die Kryptokarte geladen werden. Dazu werden jeweils mehrere Zugriffsmuster angeboten, von denen zur Laufzeit eines mittels der Werte von (nichtzufälligen) Kontrollvariablen ausgewählt wird. Die Zugriffsmuster bestimmen die Menge der Unterroutinen, die in die Kryptokarte geladen werden, und verweisen dabei ggf. auch auf Codefragmente, die nach dem Laden nicht mehr benötigt werden. Dadurch ist es für den Beobachter schwierig, anhand der Speicherzugriffe die möglichen Unterroutinen zu bestimmen, in die jetzt verzweigt werden könnte, oder funktional äquivalente Codevarianten einfach dadurch zu identifizieren, daß auf den gleichen Bereich im Hauptspeicher zugegriffen wird. Da Kryptokarte und CPU parallel arbeiten, entsteht durch die mehrfachen Speicherzugriffe solange kein Performanceverlust, wie die CPU die Vorgängerroutine abarbeitet.

In Abbildung 1 ist dargestellt, wie ein geschütztes Programm mit Hilfe der Kryptokarte abgearbeitet wird (dabei bezeichne rts den Maschinenbefehl ,,return-from-subroutine", jmp markiert einen Sprung).

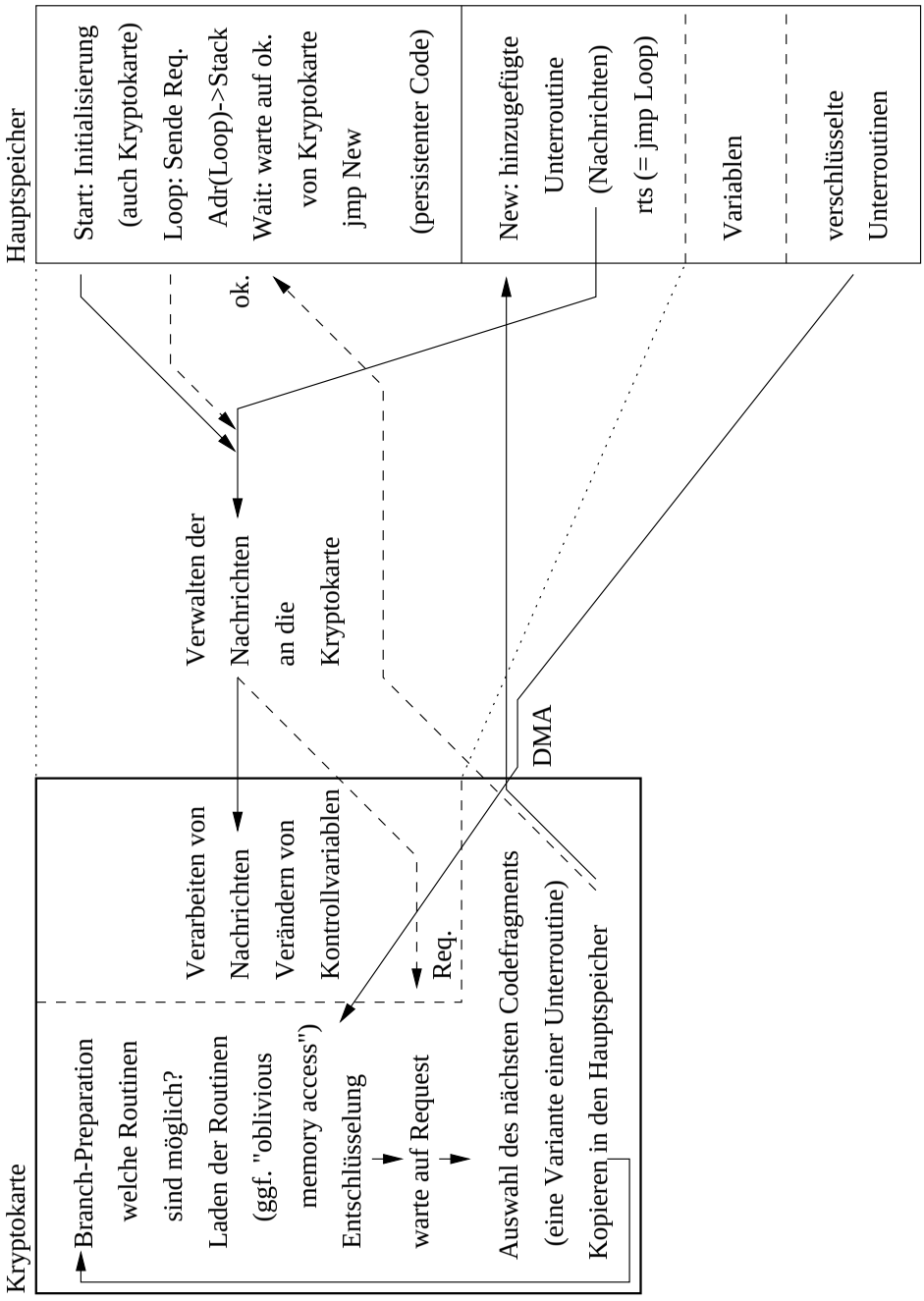


Abbildung 1: **Abarbeitung eines geschützten Programms**

2.3 Ausblick

Wenn die Kryptokarte zusätzliche Komponenten enthält, so eröffnen sich weitere Möglichkeiten bei der Zuteilung von Nutzungsrechten für Software (vgl. [11]). Mit einer (batteriegepufferten) Echtzeituhr lassen sich zeitlich befristete Lizenzen realisieren. Ist kartenintern dauerhafter Speicher vorhanden, dann können andere Wege zur Autorisierung der Nutzung von Software beschritten (zB. mit Hilfe von Smartcards [14]) oder Lizenzen auch wieder gesperrt werden. Damit wird das Übertragen von Programmen, dh. deren Weiterverkauf, möglich.

Die Kryptokarte wird in jedem Fall geheime, individuelle Schlüsselinformationen sicher verwalten und gleichzeitig gewisse kryptographische Primitive zur Verfügung stellen, zB. Verschlüsselung. Damit hat der Chip die Rolle einer ,,trusted authority", die im oben beschriebenen Fall die Nutzungsrechte für Software überprüft und zuteilt. Es bietet sich an, diese zusätzliche Hardwarekomponente für weitere Aufgaben im Bereich Systemsicherheit einzusetzen. Auch hierbei gilt es, geheime Schlüssel sicher zu verwahren und bestimmte kryptographische Funktionalitäten anzubieten. Das kann mit einer ,,tamper-resistant" Kryptokarte sicher realisiert werden, ohne von der Korrektheit der Betriebssysteme abhängig zu sein, welche als sehr komplexe Programme nie ganz fehlerfrei sein und immer wieder Einbrüche von Hackern erlauben werden. Aber selbst unter der Annahme, daß es im System gegnerische Beobachter gibt, lassen sich durch eine solche Kryptokarte zB. folgende Aufgaben relativ einfach lösen:

1. Sicheres Login, durch ein Challenge/Response-Protokoll zwischen der Kryptokarte und einer Chipkarte.

2. Erstellen von fälschungssicheren elektronischen Unterschriften. Die Kryptokarte verwahrt den dazu notwendigen private-key des Benutzers abhörsicher auf und signiert Nachrichten, falls das durch Kommunikation mit der Chipkarte autorisiert wurde.

3 Spezifikation der Kryptokarte

3.1 Kryptographische Anforderungen

Die wichtigste Maßgabe an die Kryptokarte wird sein, geheime Informationen sicher verwahren zu können. Wie diese Abhörsicherheit realisiert wird (und ob diese hundertprozentig gewährleistet werden kann), soll an dieser Stelle nicht diskutiert werden. Es gibt aber Hersteller, die Lösungen anbieten, vgl. [3] oder [15]. Um geheime Nachrichten an die Kryptokarte schicken zu können, wird diese ein Public-Key Verschlüsselungsverfahren $\mathcal{E}$ implementiert haben (Definitionen der kryptographischen Fachbegriffe finden sich zB. in [12]). Jede Karte hat ein individuell verschiedenes Schlüsselpaar (e, d); der Public-Key e ist frei verfügbar, während der Private-Key d unauslesbar in der Kryptokarte gespeichert ist. Aus Effizienzgründen wird das Public-Key Verfahren nur zum Schlüsselaustausch für ein ebenfalls vorhandenes Secret-Key Verfahren E (ein Block Cipher) dienen. Um der Kryptokarte verschlüsselte Nachrichten - in unserem Fall Codefragmente oder geheime Anweisungen an die Kryptokarte - zu übermitteln, wird zunächst via $\mathcal{E}_e(k)$ ein geheimer Schlüssel k übertragen. Dieser wird mittels der Routine `pk_decrypt`, die den Private-Key d benutzt, intern dechiffriert und unauslesbar gespeichert. Nun ist die Kryptokarte in der Lage, geheime Nachrichten $E_k(m)$ mit Hilfe des Secret-Key Verfahrens zu entschlüsseln. In unserem Szenario ergeben sich unmittelbar zwei wichtige Anforderungen an den benutzten Block Cipher:

1. Das Verfahren muß sicher gegen Known-Plaintext Attacken sein, denn während der Programmausführung wird stets ein kleiner, vormals verschlüsselter Teil des Codes unverschlüsselt im Hauptspeicher liegen.

2. Es sollte ein probabilistisches Verschlüsselungsverfahren verwendet werden, da der Nachrichtenraum sehr klein ist (Nachrichten bestehen aus Maschinenbefehlen bzw. Anweisungen an die Kryptokarte).

Die verschlüsselten Programme können beliebig kopiert werden - Backups sind kein Problem. Jeder Käufer eines geschützten Programms erhält sogar den gleichen „Ciphercode", so daß Massendistribution über CD oder Internet möglich ist. Der zum Programmstart notwendige Schlüssel k, der faktisch die Lizenz zum Ausführen des Programms darstellt, wird aber individuell verschlüsselt ausgeliefert und kann nur von der passenden Kryptokarte gelesen werden.

Um Experimente durch den Angreifer zu verhindern, muß ferner die Integrität der verschlüsselten Codefragmente und Nachrichten gewährleistet werden. Bei den Unterroutinen sind ,,destruktive" Experimente denkbar, dh. verschlüsselte Unterroutinen werden verändert bzw. zerstört (oder ausgetauscht), um herauszufinden, welche der Routinen, die in die Kryptokarte geladen wurden, letztlich zum Einsatz kommt. Um die Zuordnung, welcher Ciphertext zu welcher Klartextroutine gehört, geheimzuhalten, muß also sichergestellt werden, daß jeweils der richtige Ciphertext unverändert geladen wird. Dazu wird jede (Klartext-) Unterroutine um eine eindeutige Identifizierungsnummer und eine einfache Prüfsumme erweitert. Ein Codefragment ist dann folgendermaßen aufgebaut (im *code*-Bereich finden sich neben Maschineninstruktionen auch die in Abschnitt 2.2 erwähnten Metaregeln):

$$(id\text{-}nr, code, checksum).$$

Anschließend wird das ganze im Cipher Block Chaining (CBC-) Modus verschlüsselt, vgl. [12] und Abbildung 2. Dazu muß der Klartext in Blöcke gleicher Länge unterteilt werden, wobei der Initialblock IV beliebig, aber fest vereinbart ist. Dieser Modus hat die Eigenschaft, daß jede Verfälschung eines Ciphertextblocks bei der Entschlüsselung in einer Veränderung des zugehörigen und aller nachfolgenden Blöcke resultiert; dann stimmt (mit hoher Wahrscheinlichkeit) auch die Prüfsumme im letzten Block nicht mehr. In diesem Fall, oder wenn die erhaltene *id-nr* nicht zur angeforderten Unterroutine gehört, beendet die Kryptokarte die Abarbeitung des Programms. Stimmen dagegen *id-nr* und *checksum*, dann kann davon ausgegangen werden, daß das richtige Codefragment in unverändertem Zustand geladen wurde.

Um die Integrität der übermittelten geheimen Nachrichten sicherzustellen, werden diese ebenfalls im CBC-Modus verschlüsselt. Unter der Annahme, daß die Nachrichten mit genügend Redundanz codiert sind, resultiert jede Veränderung des Ciphertextes mit großer Wahrscheinlichkeit bei der Entschlüsselung in einer unsinnigen Nachricht (die notwendige Redundanz kann man ggf. wie oben durch Anhängen einer Prüfsumme erreichen). Weitere Experimente wie das Austauschen oder Zurückhalten von verschlüsselten Nachrichten werden durch das Abprüfen einer in die Nachricht integrierten Kontrollbedingung (die zB. durch eine Nummerierung gegeben sein kann) verhindert. Wir kommen darauf im nächsten Abschnitt zurück.

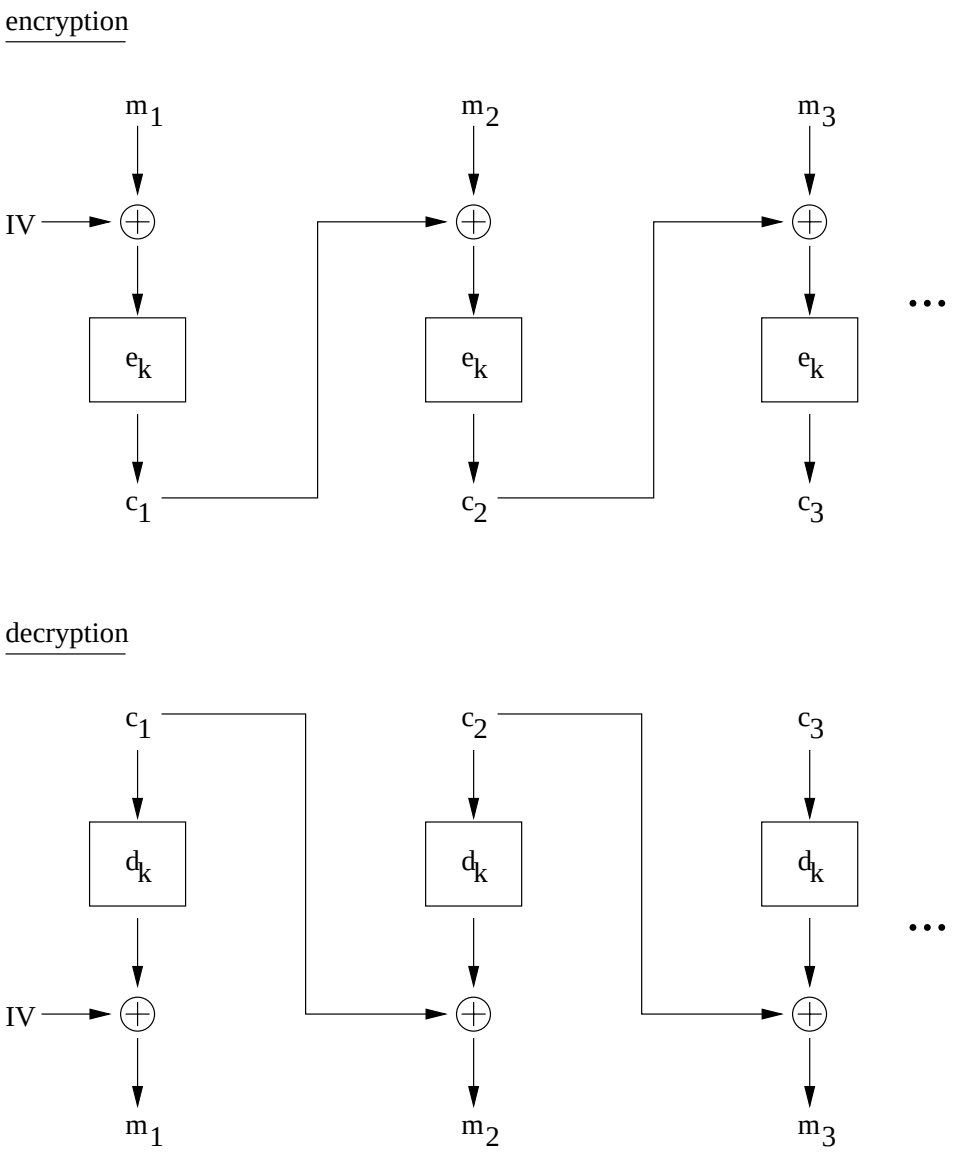


Abbildung 2: **Ver- und Entschlüsselung im CBC-Modus.**

3.2 Programmierung der Kryptokarte

Die Kryptokarte muß Entscheidungen anhand der Werte von Kontrollvariablen treffen. Dazu wird ein Interpreter benötigt für boolesche Ausdrücke über Integern, wie zB.

$$(((c_1 > 0) \text{ OR } (c_2 \neq 3)) \text{ AND } (c_1 + c_2 = 10)).$$

Ferner müssen arithmetische Ausdrücke über Integern, die als Bestandteile der booleschen Terme oder in Wertzuweisungen an Kontrollvariablen auftreten können, evaluiert werden.

Wir definieren zunächst die Sprache der Integer-Terme für die Kryptokarte (hier wirken and, or und xor bitweise):

$$\begin{aligned} op &\rightarrow +|-|*|/|\text{and}|\text{or}|\text{xor} \\ int\_expr &\rightarrow c_i|\text{Integerkonstante} \\ int\_expr &\rightarrow (int\_expr \quad op \quad int\_expr) \end{aligned}$$

Der zugehörige Interpreter ist leicht implementierbar. Im Anhang ist ein Programm angegeben, das diese Aufgabe mit möglichst wenigen Stackzugriffen löst.

`eval_int`
ZWECK: werte Integer-Terme aus
EINGABE: *int_expr*
AUSGABE: *int*
FUNKTION: siehe Anhang A.1

Boolesche Ausdrücke über Integern sind analog definiert (mit logischem AND, OR und XOR).

$$\begin{aligned} cmp &\rightarrow = | \neq | < | \leq | > | \geq \\ log &\rightarrow \text{AND}|\text{OR}|\text{XOR} \\ bool\_expr &\rightarrow true|false|(int\_expr \quad cmp \quad int\_expr) \\ bool\_expr &\rightarrow (bool\_expr \quad log \quad bool\_expr) \end{aligned}$$

Der Interpreter für diese Ausdrücke wird implementiert unter Verwendung von `eval_int`:

`eval_bool`
ZWECK: werte Bool-Terme aus
EINGABE: *bool_expr*
AUSGABE: *bool*
FUNKTION: siehe Anhang A.2

Jetzt können wir die Anweisungen an die Kryptokarte definieren, die Befehle, enthalten, Kontrollvariablen zu setzen bzw. zu verändern. Diese werden in verschlüsselten Nachrichten übermittelt, die in *Kontrollbedingung* und *Aktionsteil* untergliedert sind. Nur wenn die Kontrollbedingung erfüllt ist, wird der Aktionsteil abgearbeitet. Damit werden Experimente von Angreifern erschwert, zB. das Vertauschen oder Zurückhalten von Nachrichten, weil man leicht „Ketten" von Anweisungen bilden kann, die nur in der vorgegebenen Reihenfolge ausgeführt werden (das Verändern von Nachrichten wird durch kryptographische Techniken verhindert, vgl. Abschnitt 3.1). Im Aktionsteil werden bedingte Zuweisungen an Kontrollvariablen formuliert. Damit die Kryptokarte effizient auf Ereignisse reagieren kann, die von Programmvariablen abhängen, ist es auch möglich, die Kontrollvariablen mit dem Inhalt eines externen Registers *out_reg* zu initialisieren. Eine Zuweisung an eine Kontrollvariable c_i ist also gegeben durch

$$assign \;:=\; c_i \leftarrow int\_expr | out\_reg,$$

und eine Befehlsnachricht *order* hat folgendes Aussehen (B_i seien *bool_expr* und A_j Zuweisungen *assign*):

order	(Kontrollbed.)	if B_0 then o.k.
	(Aktionsteil)	if B_1 then $\{A_1, A_2, \ldots\}$
		elseif B_2 then $\{ \ldots \}$
		$\vdots$
		else (true) $\{ \ldots \}$

Ferner können wir nun leicht eine einfache Programmiersprache für die Kryptokarte definieren. In dieser Sprache wird die Kontrollstruktur formuliert sein. Konkret werden damit folgende Arbeitsabläufe beschrieben:

1. Zugriffe auf den Hauptspeicher, um verschlüsselten Code zu laden.

2. Auswahl der angeforderten Unterroutine.

3. Zusammenbau der aktuellen Codevariante.

Jede Programmzeile in der Kontrollstruktur enthält deshalb zwei Abschnitte *load* und *choose*, welche die ersten beiden Aufgaben steuern. Die dritte Aufgabe wird durch ein Mini-Programm *build* realisiert (entspricht den in Abschnitt 2.2 erwähnten Metaregeln), das mit dem zugehörigen Code verschlüsselt abgespeichert ist.

Im Bereich *load* werden mehrere Zugriffsmuster angeboten, von denen zur Laufzeit eines anhand der Werte von Kontrollvariablen ausgewählt wird (erneut bezeichnen B'_j boolesche Ausdrücke wie oben definiert):

load	if	B'_1	then	$\{(R_1,3),(R_7,0),(R_2,1),(R_{11},0),(R_3,2)\}$
	elseif	B'_2	then	$\{(R_2,1),(R_1,3),(R_5,0),(R_3,2),(R_6,0)\}$
	else	$(true)$	then	$\{(R_3,2),(R_5,0),(R_8,0),(R_1,3),(R_2,1)\}$

Dabei haben die Paare (R_i, j) folgende Bedeutung: lade die verschlüsselte Unterroutine $E_k(R_i)$ in die Kryptokarte. Entschlüssele diese intern und teste dabei deren Integrität wie in Abschnitt 3.1 beschrieben. Erst wenn alle im Zugriffsmuster angegebenen Routinen erfolgreich geladen, entschlüsselt und getestet wurden, fahre folgendermaßen fort (liegt eine Abweichung vor, so wurde der Ciphertext verändert; dann breche die Bearbeitung des Programms ab): falls $j \neq 0$, so speichere die Unterroutine in einem dafür vorgesehenen Speicherbereich M_j, ansonsten verwerfe den Ciphercode.

Durch das Entschlüsseln aller im Zugriffsmuster angegeben Codefragmente (auch derer, die anschließend verworfen werden), wird nicht nur deren Integrität überprüft, sondern auch verhindert, daß ein Angreifer anhand der Entschlüsselungsdauer Rückschlüsse ziehen kann auf die Menge der Unterroutinen, in die jetzt verzweigt werden könnte. Um effizient arbeiten zu können, muß die Kryptokarte also in der Lage sein, große Datenmengen schnell zu entschlüsseln. Dies erreichen wir durch parallel geschaltete Decryption-Engines,

die via Direct Memory Access (DMA) mit Ciphertextblöcken versorgt werden. Ein Scheduler verteilt dabei die vorhandenen Entschlüsselungskapazitäten sinnvoll (siehe nächsten Abschnitt).

In *choose* wird die angeforderte Unterroutine und die Zeile der Kontrollstruktur, in der fortgesetzt wird, bestimmt:

choose	if	B_1''	then	{2}	goto	{13}
	elseif	B_2''	then	{1}	goto	{15}
	else	$(true)$	then	{3}	goto	{13}

Die Interpretation ist klar: falls `eval_bool`$(B_1'') = true$, so benutze die Unterroutine, die in M_2 steht (dh. weise einer karteninternen Variablen *which* den Wert 2 zu) und setze in der Kontrollstruktur in Zeile 13 fort (dh. aktualisiere einen internen Programmzähler: $pcc \leftarrow 13$), usw.

Schließlich wird im Bereich *build*, der i.a. mehrere if-Anweisungen enthält, aus abgespeicherten Codefragmenten eine ausführbare Unterroutine zusammengebaut. Dies soll so aussehen:

build	1.	if	$(true)$	then	{0,2117}	goto	{2}
	2.	if	B_1^*	then	{2118,2330}	goto	{3}
		else	$(true)$	then	{2331,2473}	goto	{4}
	3.	if	$(true)$	then	{2474,2811}	goto	{0}
	4.	if	$(true)$	then	{2812,2978}	goto	{0}

Die Paare {a,b} bezeichnen Speicherbereiche in M_{which}, aus denen die Unterroutine zusammenkopiert wird (durch ,,goto" wird die Zeile von *build*, in der fortgesetzt wird, festgelegt; 0 = end).Die Kryptokarte soll nicht als Compiler arbeiten, sondern lediglich Codefragmente kopieren. Diese müssen evtl. noch reloziert und mit im Hauptspeicher stehendem Code gelinkt werden (Anpassen von Sprungadressen und Bekanntmachen von globalen Variablen). Dann kann der neue Code benutzt werden.

3.3 Interner Aufbau

Da die meisten modernen Betriebssysteme multitaskingfähig sind, sollte die Kryptokarte von mehreren Programmen gleichzeitig genutzt werden können.

$status$	Statusvariable, wobei 0=unused, 1=secretkey_loaded, 2=control_loaded, 3=subroutine_prepared
k	Secret-Key des Programms
$c_1, \ldots, c_n$	n Kontrollvariablen vom Typ Integer
CP	Kontrollstruktur (jede Programmzeile: $load/choose$)
pcc	Programmzähler für Kontrollstruktur
$prior$	Priorität der Task beim Entschlüsseln (0, 1 or ∞)
$succ$, $pred$	Nachfolger bzw. Vorgänger in der Queue der Tasks mit Priorität 1
$info$	Informationen, wie weit Codevorbereitung fortgeschritten, um diese nach einer Unterbrechung fortsetzen zu können
$which$	Nummer der angeforderten Unterroutine
$M_1, \ldots, M_m$	Speicherplatz für m entschlüsselte Unterroutinen

Abbildung 3: **Karteninterner Speicherbereich für eine Task**

Das Abarbeiten von geschützten Programmen durch die Kryptokarte bezeichnen wir als *Task*. Für jede Task ist kartenintern ein Speicherbereich wie in Abbildung 3 dargestellt reserviert. Durch Angabe von Prioritäten realisieren wir für das Entschlüsseln der Codefragmente ein präemptives Multitasking. Hat eine Task Priorität 0, dann hat die Kryptokarte die betreffende Unterroutine so weit wie möglich vorbereitet und wartet auf deren Anforderung (andere Tasks können abgearbeitet werden). Bei Priorität ∞ ist diese Anforderung erfolgt, die betreffende Task hat dann Vorrang vor allen anderen, damit die Wartezeit des aufrufenden Programms gering bleibt. Ansonsten teilen sich alle Tasks mit Priorität 1 die Entschlüsselungskapazitäten der Kryptokarte gleichmäßig auf. Diese Tasks werden deshalb in einer Prioritätswarteschlange verwaltet, in die bzw. aus der mit den Routinen `add_task` und `rem_task` eine Task eingefügt oder entfernt werden kann.

Die Kryptokarte muß ein entsprechendes Task-Scheduling durchführen und gleichzeitig eingehende Nachrichten abarbeiten. Letzteres wird von einer

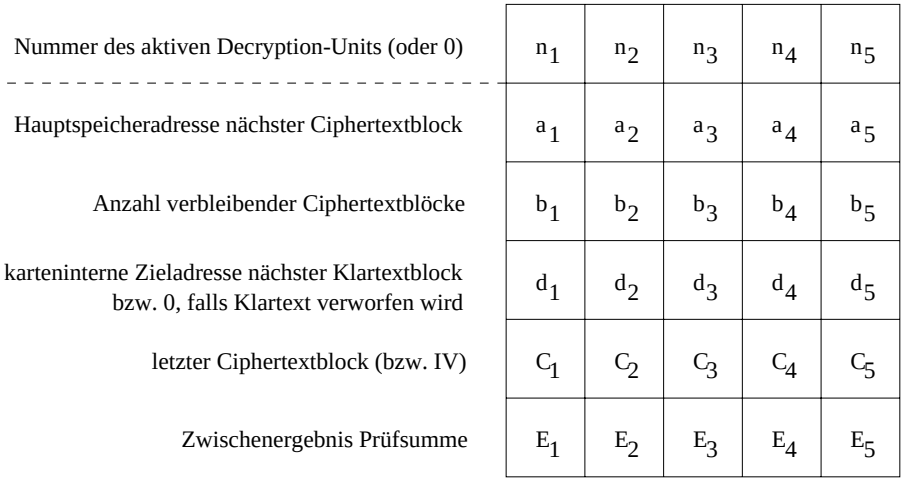


Abbildung 4: **Aufbau der** $info$**-Struktur**

Routine `driver` übernommen, welche verschiedene Schnittstellenfunktionen, die in Abschnitt 3.4 definiert werden, ausführt. Ferner gibt es eine Routine `worker`, welche sowohl die anfallende Entschlüsselungsarbeit übernimmt als auch die notwendigen Hauptspeicherzugriffe mittels DMA steuert. Normalerweise verteilt `worker` die Kapazitäten der Decryption-Units gleichmäßig an alle Tasks mit Priorität 1. Dann werden Codefragmente geladen, entschlüsselt und deren Integrität getestet (falls dieser Test negativ ausfällt, bricht `worker` die Abarbeitung des Programms ab). Diese Arbeit wird von `driver` unterbrochen, wenn eine *order* (Befehl zum Ändern von Kontrollvariablen, siehe Abschnitt 3.2) entschlüsselt und deren Integrität getestet werden muß oder eine Unterroutine angefordert wurde (setze dann die Task-Priorität auf ∞; `worker` entschlüsselt und testet ggf. noch fehlende Codefragmente und kopiert danach die angeforderte Unterroutine via DMA zurück in den Hauptspeicher). Um nach einer solchen Unterbrechung die normale Entschlüsselungsarbeit fortführen zu können, greift `worker` auf die Task-eigene Struktur $info$ zurück, deren Aufbau in Abbildung 4 angegeben ist.

3.4 Schnittstellen

Um die Kryptokarte von außen ansprechen zu können, stehen die im folgenden beschriebenen Schnittstellenfunktionen zur Verfügung. Jede Funktion repräsentiert einen Typ von Nachricht, die an die Kryptokarte geschickt werden kann. Deren Abarbeitung durch `driver` erfolgt „atomar", dh. sie kann nicht unterbrochen werden. Der zum Betrieb der Kryptokarte notwendige Treiber verwaltet daher eine FIFO-Liste (first-in-first-out) der Nachrichten, die dann sukzessive übermittelt werden.

`load_secretkey`

ZWECK:	allokiere Task und lade verschlüsselten Secret-Key
EINGABE:	$\mathcal{E}_e(k)$
AUSGABE:	Task-Nr. i bzw. 0, falls keine Task frei
FUNKTION:	if (all tasks used) then return 0 /* return = exit */
	suche freie Task i
	call `pk_decrypt`$(\mathcal{E}_e(k)) \rightarrow k(i)$
	$status(i) \leftarrow 1$ /* secretkey_loaded */
	return i

`rem_secretkey`

ZWECK:	stoppe Task und gebe Platz frei
EINGABE:	Task-Nr. i
AUSGABE:	1 = ok oder 0 = Fehler
FUNKTION:	if $(status(i) = 0)$ then return 0 /* task unused */
	if $(prior(i) = 1)$ then call `rem_task`(i)
	$status(i) \leftarrow 0$, $k(i) \leftarrow 0$
	return 1

`load_control`

ZWECK: lade verschlüsselte Kontrollstruktur CP (kann auch wiederholt geschehen) und initialisiere Task

EINGABE: Task-Nr. i, Zugriffsdaten für $E_{k(i)}(CP)$

AUSGABE: 1 = ok oder 0 = Fehler

FUNKTION: if $(status(i) = 0)$ then return 0
if $(prior(i) = 1)$ then call `rem_task`(i)
interrupt `worker`: load $E_{k(i)}(CP)$, decrypt $\rightarrow CP(i)$
$c_j(i) \leftarrow 0$ $(j = 1, \ldots, n)$ /* init. Kontrollvariablen */
$pcc(i) \leftarrow 1$, $status(i) \leftarrow 2$ /* control_loaded */
interpretiere *load*-Abschnitt der aktuellen (=ersten)
Zeile von $CP(i)$ (dabei: call `eval_bool`),
erhalte Zugriffsmuster, initialisiere $info(i)$
$prior(i) \leftarrow 1$, call `add_task`(i)
return 1

`change_variable`

ZWECK: stelle verschlüsselte Nachricht zu, anhand derer die Werte von Kontrollvariablen verändert werden

EINGABE: Task-Nr. i, verschlüsselte *order* $E_{k(i)}(B)$

AUSGABE: 1 = ok oder 0 = Fehler

FUNKTION: if $(status(i) < 2)$ then return 0 /* $CP(i)$ nicht da */
interrupt `worker`: load $E_{k(i)}(B)$, decrypt $\rightarrow B$ (=*order*)
if (Kontrollbedingung von $B = true$) then
führe Aktionsteil von B aus
(dabei: call `eval_bool` und `eval_int`)
return 1

`next_subroutine`

ZWECK:	fordere nächste Unterroutine an
EINGABE:	Task-Nr. i, Zieladresse adr für neuen Code
AUSGABE:	1 = ok oder 0 = Fehler
FUNKTION:	if ($status(i) \neq 3$) then { /* decrypt Codefrag. */
	if ($status(i) < 2$) then return 0 /* $CP(i)$ nicht da */
	if ($prior(i) = 1$) then call `rem_task`(i)
	$prior(i) \leftarrow \infty$, interrupt `worker`:
	Entschlüsseln für Task i vorrangig }
	interpretiere *choose*-Abschnitt der aktuellen Zeile der
	Kontrollstruktur (dabei: call `eval_bool`),
	setze $which(i)$, aktualisiere $pcc(i)$
	interpretiere *build*-Abschnitt der durch $which(i)$ best.
	Unterroutine, interrupt `worker`: kopiere Codefrag-
	mente via DMA in Hauptspeicher ab Adresse adr
	melde ok. /* Programm kann fortgesetzt werden */
	interpretiere *load*-Abschnitt der aktuellen Zeile von
	$CP(i)$, erhalte Zugriffsmuster, initialisiere $info(i)$
	$status \leftarrow 2$, $prior(i) \leftarrow 1$, call `add_task`(i)
	return 1

`show_publickey`

ZWECK:	fordere Public-Key der Kryptokarte an
EINGABE:	Zieladresse adr für Schlüssel
AUSGABE:	1 = ok
FUNKTION:	interrupt `worker`: kopiere e nach adr via DMA
	return 1

Abbildung 5 zeigt detailliert die internen und externen Interaktionen der Kryptokarte während der Abarbeitung eines Programms.

Abbildung 5: **Interaktionen der Kryptokarte.**

4 Erstellen geschützter Programme

4.1 Aufbereitung von Programmcode

Um Programmcode zu erzeugen, der nur mit Hilfe der Kryptokarte abzuarbeiten ist (in der oben beschriebenen Weise), bleiben dem Programmierer folgende Aufgaben:

1. Unterteilung des Programms in Stammcode, der permanent im Hauptspeicher bleibt, und in viele Wechsel-Unterroutinen, gleichzeitig Aufbau einer Kontrollstruktur und Festlegung geeigneter Nachrichten zum Verändern der Kontrollvariablen.

2. Erzeugen von Codevarianten für jede Unterroutine, effizientes Abspeichern der verschiedenen Varianten, Festlegen der Auswahlkriterien (dh. Aufbau von `build`) und der Nachrichten an die Kryptokarte.

Zur Umsetzung des ersten Punktes muß das Programm (ggf. künstlich) in viele Unterroutinen untergliedert werden. Die Unterteilungspunkte sollten an zeitunkritischen Stellen liegen, jede Unterroutine sollte eine adäquate Größe haben und - wenn möglich - zeitintensive Abschnitte beinhalten wie zB. große Schleifen, Benutzereingaben oder Peripheriezugriffe. Weiter muß eine entsprechende, komplexe Kontrollstruktur aufgebaut und eine große Menge von Nachrichten (darunter ggf. auch überflüssige) definiert werden. Es hängt vom Geschick des Programmierers ab, diese Aufgaben optimal zu lösen.

Wir zeigen nun, daß eine Untergliederung des Programms erzeugt werden kann, die eine vernünftige Abarbeitung des Codes durch die Kryptokarte ermöglicht. Jedes Programm ist aus Unterroutinen aufgebaut, die sukzessive aufgerufen werden. Für Programme, die in einer prozeduralen Hochsprache geschrieben wurden, ist das sofort klar, bei Programmen in einer objektorientierten Sprache dagegen werden häufig implizit Funktionsaufrufe erzeugt. Aber auch hier findet sich auf Maschinencode-Ebene eine Ansammlung von Subroutinen, und häufig gibt es auch Tools, die den ursprünglichen Programmcode in prozeduralen Code transformieren (so wie die frühen C++ Compiler, die lediglich Präprozessoren für C-Compiler waren).

Um eine für unsere Zwecke passende Unterteilung in Unterroutinen, die zugehörige Kontrollstruktur (die wir zunächst als Flußgraph darstellen, vgl. [4]) und notwendige Nachrichten zu erzeugen, zergliedern wir das Programm

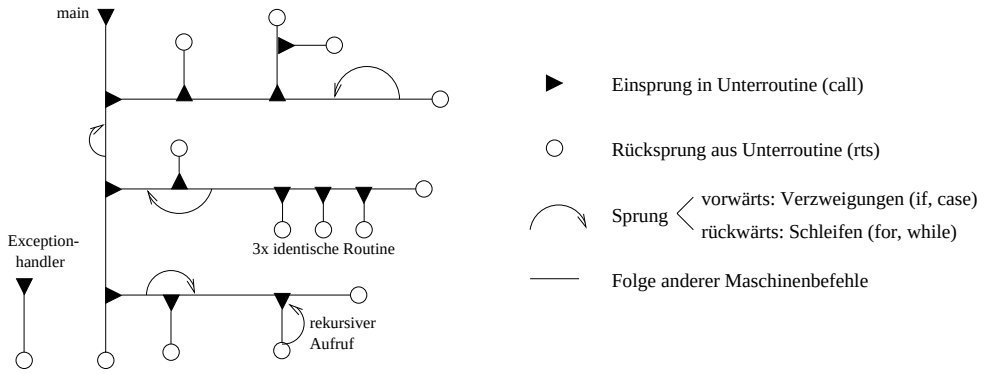
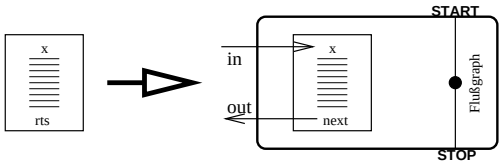


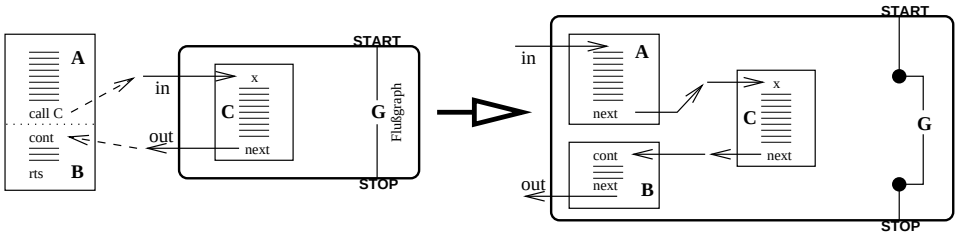
Abbildung 6: **Beispiel einer Programmstruktur.**

zuerst in sogenannte Blöcke. Routinen, die außerhalb der Aufrufstruktur stehen wie zB. Exceptionhandler, werden Teil des Stammcodes sein. Ein *Block* ist eine Codesequenz mit einem festen Einsprungpunkt *in*, einem Rücksprungpunkt *out*, sowie Aufrufparametern, Rückgabewerten und lokalen Variablen. Wir definieren Blöcke, Flußgraph und Nachrichten bottom-up aus der ursprünglichen Programmstruktur.

1. Unterroutinen, die keine weiteren Routinen aufrufen, bilden einen *primitiven* Block. Der zugehörige Abschnitt des Flußgraphen besteht aus einem Knoten, der diesen Block referenziert.

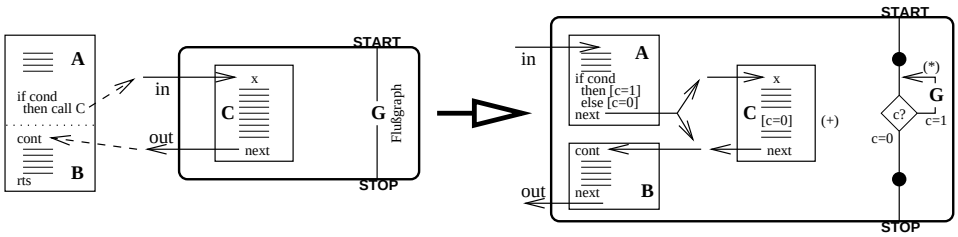


2. Routinen, die eine unbedingte Verzweigung in eine Subroutine aufweisen (darunter verstehen wir auch, daß der Aufruf nicht in einer Schleife erfolgt), welche schon in Blöcke unterteilt ist, werden auf die nachfolgend skizzierte Blockhierarchie abgebildet.



Für diesen und alle weiteren Fälle gilt: lokale Variablen, die in Teilblock A initialisiert und auch noch in Teilblock B benötigt werden, müssen als Parameter via C an B übergeben werden.

3. Bedingte Verzweigung in eine Unterroutine (hierbei bedeute [c=1], daß eine verschlüsselte Nachricht an die Kryptokarte geschickt wird, einer bestimmten Kontrollvariablen den Wert 1 zuzuweisen):



Bemerkungen

(a) Falls eine bedingte Verzweigung der Art „`if cond then {Y; call C; Z}`" vorliegt (wobei `Y` und `Z` beliebige Codesequenzen seien), dann bilden wir einen neuen Block `YCZ` (der wiederum in mehrere primitive Blöcke aufgegliedert sein kann), in den dann ggf. verzweigt wird. Daher können gleiche Codefragmente zu verschiedenen Blöcken gehören.

(b) Im Schaubild dargestellt ist die Methode, Entscheidungen *offen*, dh. im entschlüsselten Maschinencode, auszuwerten. Falls dabei ein `int`- oder `bool`-Ausdruck evaluiert wird, so kann dies auch *versteckt* in der Kryptokarte geschehen, indem Programmvariablen über *out_reg* eingeladen und Berechnungen kartenintern ausgeführt werden. Die Programmstruktur wird dadurch besser verschleiert.

(c) Auf den ersten Blick mögen die Verzweigung rückwärts (im Bild mit * markiert) und das Setzen der Verzweigungsbedingung nach „false" (+) verwundern. Dadurch läßt sich der Programmablauf in `choose` aber kompakter formulieren (siehe Beispiel am Ende dieses Abschnitts), und es wird einsichtiger, daß dieselbe Konstruktion auch für Schleifen verwendet werden kann.

```
while (cond) do            A: if (cond) then
  xxx              <=>          xxx
od                              goto A
                              fi
```

Dieses Konstrukt wird genauso umgesetzt wie oben - das `goto` wird in `choose` ausgeführt. Natürlich entfällt dabei (+), stattdessen erfolgt eine Iteration der Schleifenbedingung (offen oder versteckt).

Eine `for`-Schleife ist nur ein Spezialfall einer `while`-Schleife:

```
                           i = 1
for (i=1,...,n) do         while (i<=n) do
  xxx              <=>       xxx
od                           i = i+1
                           od
```

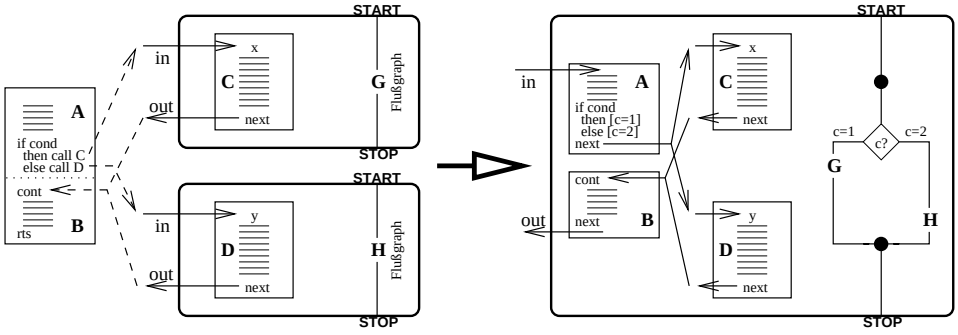
Eine `do/until`-Schleife läßt sich mit einer zusätzlichen (versteckten!) `bool`-Variablen leicht als `while`-Schleife formulieren.

```
                           exit = false
do                         while (not exit) do
  xxx              <=>       xxx
until (cond)                 if (cond) then exit = true
                           od
```

Für alle Schleifenkonstruktionen gilt jedoch: wird pro Iteration nur ein primitiver Block ausgeführt, so ist es aus Effizienzgründen sinnvoller, die Schleife in den Block zu verlagern und dort offen auszuwerten. Damit entfällt dann das wiederholte Nachladen desselben Blocks.

(d) Ebenso werden rekursive Aufrufe eines primitiven Blocks durch einen blockinternen Sprung realisiert.

4. Alternative Verzweigungen in Unterroutinen:



`switch/case`-Verzweigungen mit mehr als zwei Alternativen werden völlig analog behandelt. Bei der Programmerstellung muß allerdings darauf geachtet werden, daß es nie mehr Verzweigungsmöglichkeiten gibt als in `load` vorgesehen.

Jedes Programm läßt sich in eine solche hierarchische Blockstruktur unterteilen. Die Blöcke lassen sich (zumindest bei prozeduralen Sprachen schon auf Hochsprache-Ebene) als normale Unterroutinen darstellen; da gleiche Codefragmente zu verschiedenen Blöcken gehören können, sollte aber - um das untergliederte Programm effizient abspeichern zu können - jeder primitive Block aus einer Liste von Maschinencodesequenzen bestehen, die beim Laden zusammenkopiert werden. Beim Programmablauf werden die Blöcke von der Kryptokarte mit Hilfe der Kontrollstruktur, die sich leicht aus dem konstruierten Flußgraphen ableiten läßt, in der richtigen Reihenfolge aufgerufen. Dabei erfolgt die Übergabe von Aufrufparametern bzw. Rückgabewerten wie üblich über den Stack. Ein Teil des Codes verbleibt aber dauerhaft im Speicher. Dieser beinhaltet Initialisierungen von Kryptokarte und Hauptprogramm (Deklarationen globaler Variablen und selbstdefinierter Datenstrukturen). Weiter stehen hier Routinen außerhalb der normalen Aufrufstruktur wie zB. Exceptionhandler und ggf. kleine, sehr häufig aufgerufene Routinen, die dann aus obiger Blockhierarchie entfernt werden.

Damit ein geschütztes Programm mit Hilfe der Kryptokarte ohne größeren Performanceverlust abgearbeitet werden kann, müssen die primitiven Blöcke adäquate Größe bzw. Ausführungszeit haben. Wenn ein Block zu klein ist, so bleibt der Kryptokarte nicht genügend Zeit, den nachfolgenden Block vorzubereiten - die Ausführung des Programms gerät ins Stocken. Sehr um-

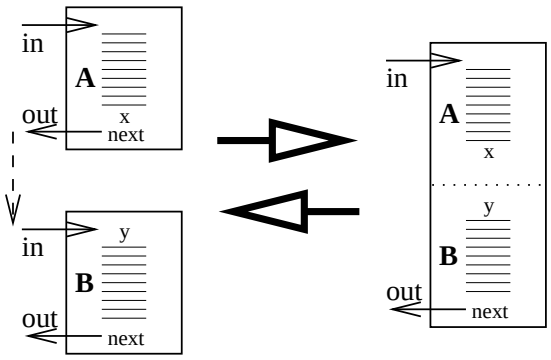


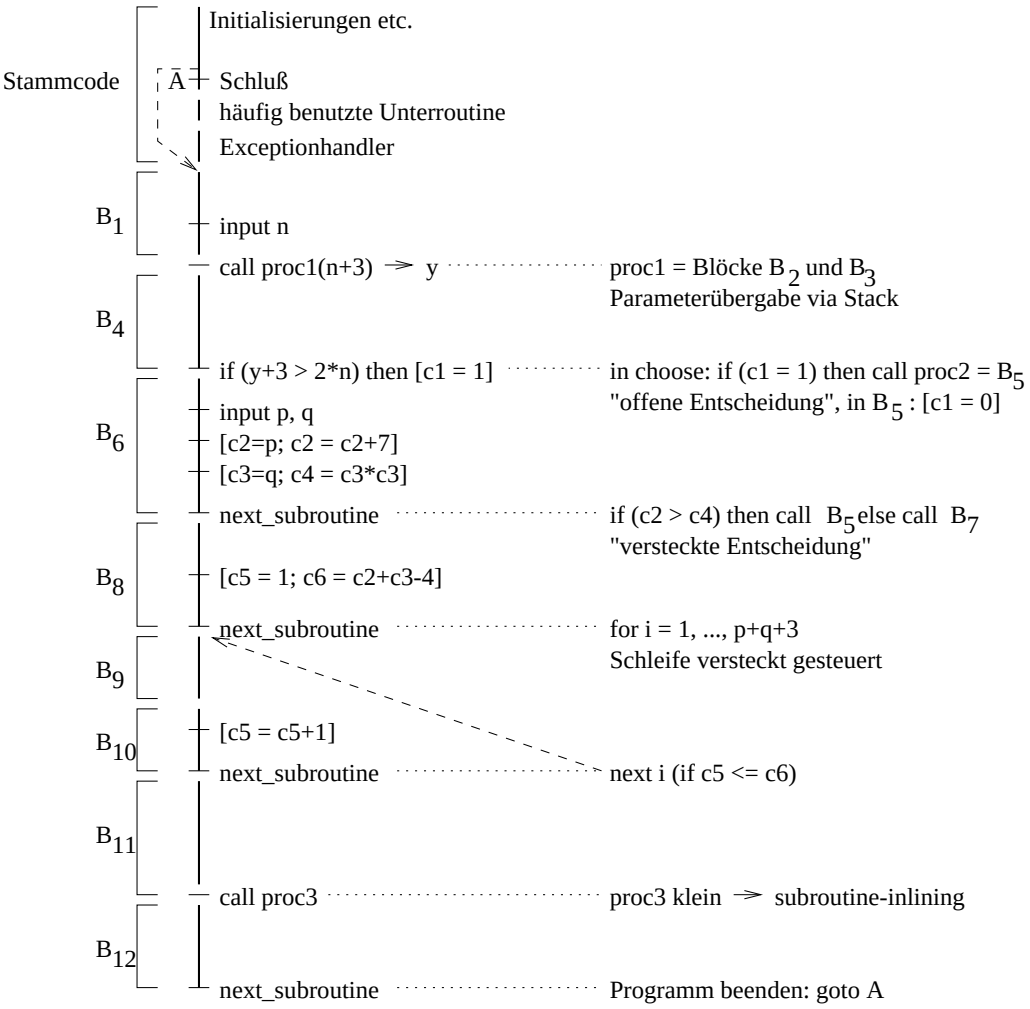
Abbildung 7: **Vergrößern bzw. Verkleinern primitiver Blöcke.**

fangreiche Blöcke brauchen wiederum viel Vorbereitungszeit, ferner ist die Blockgröße durch den zur Verfügung stehenden karteninternen Speicher begrenzt. Außerdem wird die Programmstruktur umso besser verschleiert, je mehr Blöcke es gibt. Ausgehend von der oben beschriebenen Blockhierarchie eines Programms können primitive Blöcke leicht vergrößert bzw. verkleinert werden, vgl. Abbildung 7. Dabei können Aufrufparameter zu lokalen Variablen werden oder umgekehrt.

Blöcke verkleinert man dadurch, indem sie künstlich in Teilblöcke unterteilt werden. Das Vergrößern erfolgt, indem aufeinanderfolgende Blöcke zu einem einzelnen zusammengefaßt werden. Ist in `choose` angegeben, daß nach Block *A* nur eine einzige Fortsetzung möglich ist (so wie in Abbildung 7 angedeutet), so entspricht dies einfach „Subroutine-Inlining", vgl. [4]. Sind dagegen mehrere Nachfolgeblöcke möglich, dann muß ferner die Verzweigungsbedingung offen im Programm ausgewertet werden; im Programmcode sind dann alle Alternativen enthalten. Geschieht dieses Zusammenfassen wiederholt, so kann es vorkommen, daß der gleiche Block mehrfach übernommen werden müßte. Dann ist es allerdings sinnvoller, diesen nur einmal als blockinterne Unterroutine einzubinden.

Wir erläutern nun das Konzept an einem kleinen Beispiel.

Exemplarisches „Programm"



Die zugehörigen `choose`-Anweisungen:

```
(1) if   (true) then B1  goto  2
(2) if   (true) then B2  goto  3
(3) if   (true) then B3  goto  4
(4) if   (true) then B4  goto  5
(5) if   (c1=1) then B5  goto  5
    else (c1=0) then B6  goto  6
```

```
 (6) if  (c2>c4) then B5  goto  7
     else (true) then B7  goto  7
 (7) if   (true) then B8  goto  8
 (8) if (c5<=c6) then B9  goto  9
     else (true) then B11 goto 10
 (9) if   (true) then B10 goto  8
(10) if   (true) then B12 goto  0
```

Um zur Laufzeit das dynamische Einfügen der Unterroutinen in das aktive Codesegment zu erleichtern, sollte zunächst beim Übersetzen des Programms relozierbarer Code erzeugt werden. Dies geschieht normalerweise durch einfaches Setzen einer Compileroption, welche bewirkt, daß alle Adressierungen relativ zum Programmzähler angegeben werden. Damit entfällt das Anpassen von Adressen, wenn eine Routine nachgeladen wird. Auch das dabei notwendige Anbinden des neuen Codes an die Stammroutine kann effizient gestaltet werden. Eine erste Möglichkeit ist, die Unterroutinen als eigenständige Module zu übersetzen und diese zur Laufzeit *dynamisch* zu linken, vgl. [1] (für viele Betriebssysteme sind dynamische Linker verfügbar, zB. `dld` für UNIX [16]). Das wird nicht zu zeitaufwendig sein, denn die Unterroutinen sind so gestaltet, daß nach deren Anbinden an die Stammroutine keine weiteren Module benötigt werden, dh. der Linkprozess setzt sich nicht rekursiv fort. Eine effizientere Strategie aber ist „Prelinking", dh. das Programm wird übersetzt (relozierbar und ggf. mit Debug-Informationen) und normal statisch gelinkt. Die resultierende ausführbare Datei wird anschließend wieder in die einzelnen Unterroutinen zergliedert, die dann zur Laufzeit ohne einen weiteren Linkvorgang nachgeladen werden können.

4.2 Erzeugen von Codevarianten

Das Erstellen von Varianten für jede Unterroutine kann auf Assemblerebene problemlos geschehen. Aber auch in der verwendeten Hochsprache können die Unterroutinen variabel definiert werden. Jede Variante durchläuft dann den oben beschriebenen Prozess des Übersetzens und ggf. des Prelinkens. Die so entstandenen Codevarianten können kompakt abgespeichert werden, indem eine `build`-Struktur aufgebaut wird, dh. gleiche Codefragmente, die in verschiedenen Varianten vorkommen, werden nur einmal abgespeichert. Das Erzeugen von geeigneten Kontrollanweisungen zur Auswahl der aktu-

ellen Variante sollte ebenfalls kein Problem darstellen. Bei der Definition polymorpher Unterroutinen ist der Einfallsreichtum des Programmierers gefordert. Hier einige Möglichkeiten, funktional äquivalente Codevarianten zu erzeugen:

- Dummy-Befehle, die auf Dummy-Variablen operieren.
- Schleifen rückwärts statt vorwärts durchlaufen.
- Direkte Adressierung durch indirekte ersetzen.
- Fehlerpropagierung und -korrektur.
  Bsp: zur Berechnung von $a \rightarrow a^2$ erzeuge Routine für $x \rightarrow x^2 - 2x - 3$, rufe diese mit Parameter $a + 1$ auf und korrigiere den Rückgabewert $b \leftarrow b + 4$.
- Flexible Unterteilung des Programms, Verschieben von Blockgrenzen.
- Einführen von zusätzlichen Unterroutinenparametern. Benutze diese für fingierte bedingte Verzweigungen, um toten Code zu erzeugen, der natürlich beliebig variiert werden kann.
- Erzeuge nichtoptimierten statt optimierten Code durch Compiler.
- Veränderungen auf Maschinenebene, zB. Permutation von Registern.

5 Schlußbemerkungen

Wir haben ein neues Konzept zur Realisierung von Softwarekopierschutz vorgestellt. Unser Ansatz ist aufwendiger als die bisher vorgeschlagenen, dafür scheint eine Umsetzung ohne allzu großen Performanceverlust möglich, vor allem, weil die Entschlüsselung von Code bzw. Nachrichten kaum noch Zeitverlust mit sich bringen muß. Bei der konkreten Realisierung wäre vor allem wichtig, die Hauptspeicherzugriffe der Kryptokarte - lesende und schreibende - effizient und sicher durchzuführen, zB. durch (mit dem Cachesystem verträgliches) DMA. Die Kryptokarte selbst benötigt relativ viel internen Speicher, um jeweils mehrere Unterroutinen in verschiedenen Varianten abspeichern zu können. Unserer Meinung nach kann das Konzept am besten bei großen Programmen angewendet werden, ganz besonders bei solchen, die sowieso dynamisch gelinkte Unterroutinen benutzen.

Literatur

[1] L. L. Beck. *System Software: An Introduction to Systems Programming*, 2nd edition, 1990.

[2] R. M. Best. *Microprocessor for Executing Enciphered Programs*, US Patent No. 4 168 396, 1979.

[3] Concord Eracom und ERACOM. *Sicherheitsprodukte und Komponenten*, Produktinformation 1995.

[4] K. Dowd, *High Performance Computing*, 1993.

[5] O. Goldreich, R. Ostrowski. *Software Protection and Simulation on Oblivious RAMs*, Technical Report TR-93-072, University of California at Berkeley, 1993.

[6] D. Grover (ed.). *The Protection of Computer Software*, Monographs in Informatics, British Computer Society, 1989.

[7] A. Herzberg, S. Pinter. *Public Protection of Software*, Advances in Cryptology - CRYPTO'85 Proceedings, pp. 158-179.

[8] S. T. Kent. *Protecting Externally Supplied Software in Small Computers*, Ph. D. Thesis, MIT/LCS/TR-255, 1980.

[9] D. Keppel. *A Portable Interface for On-The-Fly Instruction Space Modification*, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV) 1991, pp. 86-95.

[10] B. Krämer, M. Kiel. *Harte Nüsse geknackt, Kopierschutzsysteme näher betrachtet*, c't 5/1990.

[11] G. B. Purdy, G. J. Simmons, J. A. Studier. *A Software Protection Scheme*, 1982 IEEE Symposium on Security and Privacy - Proceedings, pp. 99-103.

[12] B. Schneier. *Applied Cryptography*, 2nd edition, 1995.

[13] VLSI Technology Inc. *Data Encryption Products*, Product Bulletin, 1992.

[14] S. R. White, L. Comerford. *ABYSS: A Trusted Architecture for Software Protection*, 1987 IEEE Symposium on Security and Privacy - Proceedings, pp. 38-51.

[15] S. R. White, S. H. Weingart, W. C. Arnold, E. R. Palmer. *Introduction to the Citadel Architecture: Security in Physically Exposed Environments*, IBM Research Division - Research Report RC 16672, 1991.

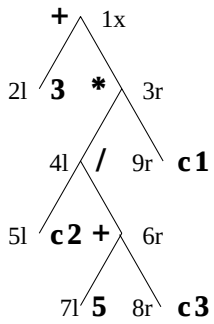
[16] W. Wilson Ho, R. A. Olsson. *An Approach to Genuine Dynamic Linking*, Software Practice and Experience, Vol. 21 (4), 1991, pp. 375-390.

A Realisierungen

A.1 Interpreter für Integerterme

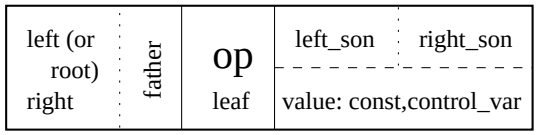
Wir repräsentieren einen Integerterm als Binärbaum (a), der intern durch folgende Datenstruktur dargestellt wird: jeder Knoten ist eine Struktur (b), der Baum wird auf ein Array dieser Records abgebildet (c).

a)



(3+((c2/(5+c3))*c1)

b)



c)

1				
l	0	+	2	3

2			
l	1	leaf	3

3				
r	1	*	4	9

4				
l	3	/	5	6

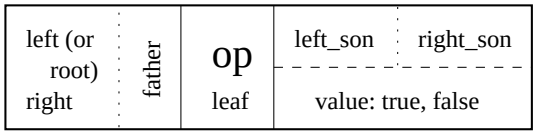
Da zu jedem Knoten der Vater und die eigene Position (links/rechts) bekannt sind, müssen beim Auswerten des Baums lediglich die Zwischenergebnisse der linken Knoten auf einem Stack gespeichert werden. Damit wird die Anzahl der Stackzugriffe minimiert.

```
procedure eval_int(array A)
{ act  = A[0];
  exit = false;
  do  { if (act.op != leaf)  /* do-until */
        then { act = act.left_son; }
        else { res = act.value;    /* const|control_var */
               while (act.pos == right) do
                  { act = act.father;
                    aux = pop_stack();
                    res = (aux act.op res); }  /* while */
```

```
                if (act.father != 0)  /* act.pos == left */
                 then { push_stack(res);
                        act = act.father.right_son; }
                 else { exit = true; }
                } /* else */
  } until exit;
  return res;
} /* eval_int */
```

A.2 Interpreter für boolsche Terme

Jeder boolsche Term wird ebenfalls durch einen Binärbaum repräsentiert, dessen Blätter von der Form *true* oder *false* sind. Diesen Baum speichern wir als Array von Strukturen, die folgendes Aussehen haben.



Das ist das gleiche Format, das wir für Integer-Ausdrücke benutzen. Wir können deshalb auch `bool`-Bäume, die als Teilbäume `int`-Ausdrücke besitzen, kompakt in einem Array abspeichern. Anbhängig vom *op*-Eintrag wird ggf. die Evaluierungsroutine `eval_int` aufgerufen. Da der Baum zweifach verkettet abgespeichert ist, kann wiederum die Anzahl der Stackzugriffe minimiert werden.

```
procedure eval_bool(array A)
{ act  = A[0];
  exit = false;
  do  { if (act.op == log)  /* do-until */
          then { act = act.left_son; }
          else { if (act.op != leaf)     /* (int cmp int) */
                  then { a = eval_int(act.left_son);
                         b = eval_int(act.right_son);
                         res = (a act.op b); }
                  else { res = act.value; } /* true|false */
                 while (act.pos == right) do
```

```
                    { act = act.father;
                      aux = pop_stack();
                      res = (aux act.op res); }  /* while */
                  if (act.father != 0)  /* act.pos == left */
                   then { push_stack(res);
                          act = act.father.right_son; }
                   else { exit = true; }
                  } /* else */
  } until exit;
  return res;
} /* eval_bool */
```