Fortgeschrittene Techniken zur sicheren Ausführung mobilen Codes durch Obfuscation

Baccalaureatsarbeit

vorgelegt von Stefan Reich

betreut durch Prof. K. Brunnstein

Arbeitsbereich AGN

Fachbereich Informatik

Universität Hamburg

2. 10. 2001

Inhaltsverzeichnis

1 Einleitung.....5

1.1 Problemstellung, Szenario.....5

1.2 Realisierbarkeit.....5

1.3 Überblick.....6

1.4 Anwendungsbeispiele.....6

1.4.1 Verteilte Berechnungen im Internet.....6

1.4.2 Online-Spiele.....7

2 Grundlegendes Verfahren.....8

2.1 Definitionen.....8

2.1.1 Code, mobiler Code.....8

2.1.2 Server, Client.....9

2.1.3 Kritische Information.....9

2.1.4 Obfuscation.....9

2.2 Grundidee.....10

2.3 Security through obscurity?.....11

3 Aktuelle Implementation.....13

3.1 Komponenten.....13

3.2 Der Rewriter.....14

3.3 Protokoll.....16

3.3.1 Ereignisse.....17

3.3.2 Parametrische Ereignisse.....17

3.3.3 Server-Antworten.....18

3.3.4 Variationen des Protokolls.....18

3.4 API.....19

3.4.1 drjava.marvin.api.Talkback.....19

3.4.2 drjava.marvin.api.Hook.....20

3.4.2.1 Schnittstelle.....20

3.4.2.2 Beschreibung.....20

3.4.2.3 Methoden im Detail.....21

3.4.2.4 Verwendung im Kontext.....23

4 Verbesserungen der Implementation....................24
4.1 Stringliterale....................24
4.1.1 Einführung....................24
4.1.2 Aktuelle Situation....................24
4.1.3 Verbesserungsvorschlag....................25
4.1.4 Umsetzung....................26
4.1.5 Bewertung....................28
4.2 Dynamische Kodierung der kritischen Information....................28
4.2.1 Aktuelle Situation....................28
4.2.2 Verbesserungsvorschlag....................29
4.2.3 Umsetzung....................30
4.2.3.1 API-Erweiterungen....................30
4.2.3.2 Kodierung....................30
4.2.3.3 Protokoll....................31
4.2.4 Bewertung....................31
4.3 Verstecken von Referenzen auf Systemklassen....................32
4.3.1 Einführung....................32
4.3.2 Aktuelle Situation....................34
4.3.3 Verbesserungsvorschlag....................34
4.3.4 Umsetzung....................35
4.3.5 Beispiel....................35
4.3.6 Bewertung....................36
4.4 Code Streaming....................37
4.4.1 Aktuelle Situation....................37
4.4.2 Verbesserungsvorschlag....................37
4.4.3 Umsetzung....................38
4.4.4 Bewertung....................39
5 Gesamteinschätzung des verbesserten Systems....................41
5.1 Bisherige Schwachpunkte....................41
5.2 Erreichte Fortschritte....................41
5.3 Verbleibende Risiken....................42
6 Referenzen....................44
6.1 Verwandte Arbeiten....................44

6.2 Literatur..44
6.3 WWW...45

1 Einleitung

1.1 Problemstellung, Szenario

Im allgemeinen wird bei der Diskussion um Sicherheit im Internet und bösartigen Programmcode die Sicht des Nutzers, also im Client/Server-Modell des Clients, eingenommen. Es werden Vorschläge gemacht, wie sich Nutzer davor schützen können, Inhalte von nicht vertrauenswürdigen Servern herunterzuladen, die auf dem eigenen Rechner Schaden anrichten würden.
Zweifellos ist dies ein wichtiges Teilgebiet der IT-Sicherheit.
In dieser Arbeit soll allerdings ein anderer Blickpunkt eingenommen werden. Der Betreiber eines Servers wird als vertrauenswürdig angenommen, und wir stellen die Frage, wie der Server mit möglicherweise böswilligen Clients kommunizieren und dennoch Annahmen über die Integrität der eingehenden Daten machen kann.
Konkreter sieht das Szenario so aus: Der Betreiber eines Servers liefert Programmcode zur Ausführung an u.U. nicht vertrauenswürdige Clients und möchte dabei sicherstellen, dass dieser Code während der Ausführung vor Manipulationen geschützt ist. Mindestens sollen solche Manipulationen anhand der Client-Server-Kommunikation erkennbar sein.

1.2 Realisierbarkeit

Von einem theoretischen Standpunkt aus ist dieses Vorhaben zum Scheitern verurteilt. Clients sind inhärent nicht vertrauenswürdig, da keine Annahme darüber gemacht werden kann, wie die clientseitig installierten Programme, mit denen der Server kommuniziert (Web-Browser, Appletviewer, Virtual Machine, Betriebssystem) beschaffen sind. Hierüber hat nur der Administrator des Clientrechners die Kontrolle.
Setzt man allerdings die Messlatte etwas niedriger und fordert nicht die Unmöglichkeit eines Betrugs, sondern nur, dass jeder denkbare Angriff zeitlich sehr aufwändig sein und hohe Fachkenntnis und Intelligenz seitens des Angreifers erfordern muss, läßt sich ein Arsenal von Maßnahmen zusammenstellen, das in seiner Kombination sehr effektiven Schutz verspricht.

1.3 Überblick

In der vorliegenden Arbeit wird ein Maßnahmenkatalog zum Schutz von Java-Applets vorgestellt und diskutiert. Kapitel 2 erläutert das zu Grunde liegende Verfahren und definiert zentrale Begriffe. Kapitel 3 beschreibt die existierende Implementation des Verfahrens. Im 4. Kapitel werden Schwächen der aktuellen Implementation und mögliche Verbesserungen diskutiert. Die aus dieser Diskussion enstandenen Ansätze zur Modifikation des Systems werden jeweils konkret umgesetzt und bewertet. Das 5. Kapitel schließlich versucht eine Gesamtbewertung der geleisteten Arbeiten.

1.4 Anwendungsbeispiele

1.4.1 Verteilte Berechnungen im Internet

Für Berechnungen, die zu aufwändig für eine einzelne Workstation oder gar einen Supercomputer oder einen Cluster sind, etabliert sich seit einer Weile ein neues Modell: Auf einer öffentlichen Website werden alle Internetnutzer eingeladen, die Rechenleistung ihres Privat- oder Büro-PCs zur Verfügung zu stellen.
Die Teilnehmer laden dann eine spezielle Clientsoftware herunter, die sich regelmässig Datenpakete von einem zentralen Server lädt, diese bearbeitet und das Ergebnis zurücksendet.
Konkrete Bespiele sind SETI@Home [Suche nach Ausserirdischen, SET] , FightAIDS@Home [Simulation der Wirkung von Medikamenten, FIG].
Die Nutzer sind in diesem Modell aus Sicht des Initiators im allgemeinen nicht vertrauenswürdig. Es besteht die Gefahr, dass die von Clients gelieferten Ergebnisse nicht authentisch sind.
SETI@Home veröffentlicht zum Beispiel Ranglisten der aktivsten Teilnehmer. Weit oben auf diesen Listen zu erscheinen ist für einige offenbar ein Anreiz, die Bearbeitung der Datenpakete nur zu simulieren statt wirklich durchzuführen. Gerade bei SETI stellte sich das als gravierendes Problem heraus, weil es keinen effizienten Algorithmus gab, um die Korrektheit von Ergebnissen zu prüfen.

Diese Anwendungen könnten von einem System zur sicheren Ausführung mobilen Codes enorm profitieren. Begleitet von weiteren Maßnahmen kann dies zu sehr hoher Verlässlichkeit der Berechnungsergebnisse führen.
Ein Hindernis bei der Anwendung des vom Autor entwickelten Systems ist in diesem Kontext allerdings, dass aufwändige Berechnungen aus Performancegründen im allgemeinen nicht in Java implementiert werden. Es spricht prinzipiell jedoch nichts dagegen, das Funktionsprinzip auf andere Sprachen, etwa C, zu übertragen.

1.4.2 Online-Spiele

Interaktive, schnelle Online-Spiele müssen im wesentlichen auf dem Clientrechner ablaufen, weil die Latenz der Verbindung meist zu hoch ist für eine Kommunikation bei jedem Spielschritt. Für Action- und Geschicklichkeitsspiele gilt dies in jedem Fall. Oft werden bei Online-Spielen Bestenlisten geführt; nicht immer nur zur Ehre der Spieler – teilweise werden reale Preise für die höchsten Punktzahlen ausgelobt.
Hier stellt sich ein ähnliches Problem wie im Beispiel der verteilten Berechnungen – es gibt eine Motivation für unlautere Teilnehmer, sich durch „Hacken" einen Platz in der Liste zu sichern. Die Betreiber eines Onlinespiels sind daran interessiert, dies zu unterbinden, da es auf lange Sicht die Attraktivität ihres Angebots schädigt.
Eine Website, die solche Spiele anbietet, ist z.B. Gambas [GAM]. Dort gibt es sowohl Ranglisten wie Gewinne – bei relativ geringem Schutz vor Hackern.
Die Anwendung, die mich überhaupt zu dem in dieser Arbeit vorgestellten Projekt inspirierte, ist ebenfalls ein Online-Spiel – eine Flippersimulation, um genau zu sein [PHA]. Der Flipper ist über das Internet frei spielbar und wird seit etwa einem Jahr von einem Obfuscation-basierten Schutzsystem ausgeliefert. Preise für Highscores werden allerdings nicht vergeben.

2 Grundlegendes Verfahren

2.1 Definitionen

In dieser Arbeit werden einige Begriffe in einer spezialisierten Bedeutung benutzt; daher werde ich sie im folgenden definieren und auf Unterschiede zu anderen gebräuchlichen Verwendungen hinweisen.

2.1.1 Code, mobiler Code

Das in unserem Kontext wesentliche Artefakt ist der Programmcode, hier kurz *Code*.

> Def. **Code**: Maschinenlesbar kodiertes, ausführbares Programm oder Programmstück; kann in nativer Maschinensprache (z.B. x86), in maschinenunabhängigem Pseudocode (z.B. Java-Bytecode) oder im Quelltext vorliegen (Beispiel Skriptsprachen)

Dies ist eine sehr allgemeine und gängige Definition. Der Begriff des *mobilen Codes* wird hier dagegen in einer speziellen Auslegung verwendet:

> Def. **Mobiler Code**: Code, der unmittelbar vor der Ausführung durch einen Kommunikationskanal zum ausführenden Rechner übertragen wird. Übermittlung und Ausführung müssen konzeptuell ein integrierter, automatisierter Vorgang sein (im Gegensatz beispielsweise zu einem Programmdownload mit anschliessender manueller Ausführung).

Andere Auslegungen dieses Begriffs schliessen vor allem auch das mehrfache Übertragen des Codes zwischen mehr als zwei Rechnern ein. Eine andere Variante ist die *Migration* mobiler Agenten; hier wird die Ausführung des Codes angehalten, Code und aktueller Zustand werden serialisiert, übertragen und die Ausführung in einer anderen Umgebung fortgesetzt.

2.1.2 Server, Client

Def. **Server**: Computer, der mobilen Code aussendet (genauer: das Programm auf diesem Computer, das dafür zuständig ist)
Def. **Client**: Computer, der mobilen Code empfängt und ausführt (genauer: das Programm, das dafür zuständig ist)

Auch diese beiden Bezeichnungen wurden speziell auf den Kontext hin definiert.

2.1.3 Kritische Information

Def. **Kritische Information**: Ergebnis einer clientseitigen Berechnung oder Benutzerinteraktion, das zum Server übertragen wird; das Ziel der sicheren Codeausführung ist, sicherzustellen, dass die kritische Information, die den Server erreicht, authentisch ist, also vom unverändert ausgeführten mobilen Code erzeugt wurde.

Die Bezeichung *Kritische Information* ist meines Wissens eine Neuschöpfung. Er soll der Unterscheidung der Wichtigkeit verschiedener Informationen dienen, die zwischen Client und Server ausgetauscht werden.
Allen betrachteten Anwendungen ist gemeinsam, dass zum Ende des Vorgangs, oder auch mehrfach während des Ablaufs, bestimmte Datenpakete übermittelt werden, die von besonderer Bedeutung sind. Diese fliessen immer vom Client zum Server und enthalten die eigentlich „interessierende“ Information; beispielsweise das Ergebnis von clientseitigen Berechnungen.
Diese Information wird im im Protokoll besonders behandelt; es ist daher zweckmässig, einen eigenen Begriff einzuführen.

2.1.4 Obfuscation

Def. **Obfuscation**: von „to obfuscate“ (verfinstern, verdunkeln, verwirren); funktionserhaltende Transformation von Programmcode mit

dem Ziel, Analyse und Verständnis der Funktionsweise seitens Dritter zu erschweren. In dieser Arbeit wird synonym das eingedeutschte **obfuszieren** bzw. **Obfuskation** verwendet.

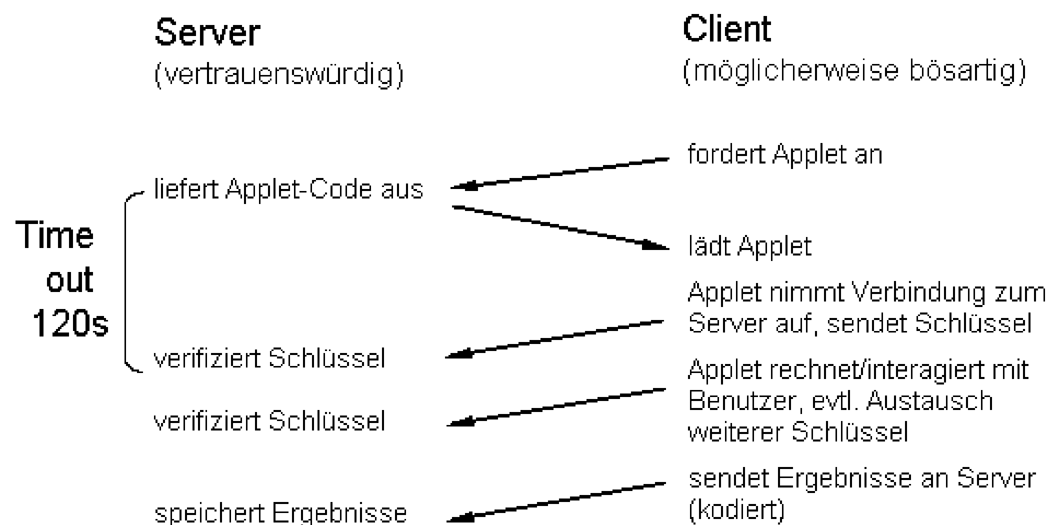
Ein verwandter Terminus ist *Polymorphismus*, oder *polymorpher Programmcode*. Dieser wird vor allem im Bereich der Malware, insbesondere bei Viren, verwendet. Viele Viren sind in der Lage, beim Reproduzieren die Struktur ihres eigenen Codes zu verändern; wiederum natürlich funktionserhaltend, daher kann man auch hier von Obfuskation sprechen. Die Zielsetzung ist leicht verändert, da nicht die Analyse, sondern die Identifikation eines bestimmten Prorgammcodes verhindert werden soll; die Mittel aber sind sehr ähnlich.
Es gibt zwei grundlegende Zwecke, zu denen Obfuskation üblicherweise eingesetzt wird. Zum einen ist dies der Schutz des geistigen Eigentums, das in den Algorithmen und Methoden eines Programms steckt. In einer Wettbewerbssituation sehen Softwarefirmen die genaue Funktionsweise ihrer Produkte als Betriebsgeheimnis an und versuchen diese daher zu verbergen. Zum zweiten dient Obfuskation dem Bestreben, Programmcode vor unauthorisierter Veränderung zu schützen, beispielsweise um das Aushebeln von Kopierschutzmaßnahmen zu verhindern.

2.2 Grundidee

Das hier vorgestellte Verfahren basiert auf vier Prinzipien:

1. Der mobile Code wird grundsätzlich obfusziert ausgeliefert.
2. Die Obfuskation wird vor jeder Auslieferung erneut vorgenommen; Zufallsparameter bewirken dabei jedes Mal ein unterschiedliches Ergebnis.
3. Das Client-Server-Protokoll wird für jede Auslieferung neu definiert und in den mobilen Code eingebettet.
4. Die erlaubte Zeitspanne zwischen Code-Auslieferung und Rückmeldung des Clients ist auf einen festen Wert t_{max} begrenzt.

Diese Maßnahmen sollen die primäre Angriffsstrategie unterlaufen: die Ausführung manipulierten Codes anstelle des ausgelieferten. Der Voraussetzung hierfür - Reverse Engineering, Analyse und gezielte Modifikation - wird durch die Kombination von (3) und (4) der Boden entzogen. Da jede Inkarnation des mobilen Codes ein „Verfallsdatum“ besitzt, muss die gesamte Analyse und Modifikation innerhalb von t_{max} erfolgen; zudem noch abzüglich Übertragungs- und Ausführungszeiten.
Je komplexer der ursprüngliche Code, die Protokollvariation und die Obfuskationstechniken angelegt sind, desto schwieriger wird es, diese Aufgabe innerhalb von t_{max} zu bewältigen. Um einen groben Anhaltspunkt zu geben: t_{max} kann in den meisten Fällen in den zweistelligen Sekundenbereich gelegt werden.
Der schematische Ablauf im Zeitdiagramm, t_{max} = 120s:



2.3 Security through obscurity?

Im Rahmen des Vortrags über das in dieser Arbeit vorgestellte System wurde von Zuhörern die Frage gestellt, ob es sich nicht um ein Beispiel von *security through obscurity* handle; also um einen Versuch, Sicherheit nur dadurch herzustellen, dass die verwendeten Mechanismen nicht öffentlich bekannt sind.
Security through obscurity wird als Gegensatz gesehen zu „wirklicher“ Sicherheit. Wirklich sichere Systeme sind dies auch dann noch, wenn ihre Funktionsweise

dokumentiert und veröffentlicht ist – beispielsweise durch eine Open-Source-Implementation.

Zunächst eine Vorbemerkung: Wie in 1.2 bereits festgestellt, kann aus der Sicht der Theorie kein Verfahren, dass Programmcode vor dem Adminstrator des ausführenden Rechners „schützen“ soll, wirklich funktionieren, da dieser im Prinzip volle Kontrolle über alle Aktionen der Maschine hat. Praktisch gesehen sind Menschen aber Maschinen in mancherlei Hinsicht unterlegen, vor allem in bezug auf ihre Reaktionsgeschwindigkeit; vor allem letzteres stellt die Basis dar für die Idee des hier diskutierten Verfahrens.

Was würde geschehen, wenn die Quelltexte dieses Systems vollständig offen gelegt würden? Nun – ein Teil der Arbeit, die ein Angreifer zu leisten hat, wird vereinfacht. Er kann sich anhand der Quelltexte ein vollständiges Bild davon machen, welche Obfuskationsmethoden und Protokollvariationen der Server einsetzt. Hat er damit bereits die Mittel in der Hand, einen konkreten Server erfolgreich anzugreifen? Nein – denn nach wie vor gilt es, innerhalb der vom Server gesetzten Zeitbeschränkung das ausgelieferte Applet zu analysieren und zu modifizieren. Auch mit dem Hintergrundwissen über die Funktionsweise des Systems ist dies manuell praktisch nicht zu bewältigen.

Eine zusammenfassende Antwort kann also etwa lauten: Das Verfahren fällt im Prinzip in die Kategorie Security by obscurity; allerdings ist die Entwicklung eines *Cracks* auch bei voller Kenntnis des Verfahrens alles andere als trivial, da dafür eine komplexe automatische Codeanalyse implementiert werden muss.

3 Aktuelle Implementation

Die Implementation der in Kapitel 2 vorgestellten Idee wurde vom Autor im Sommer 2000 entwickelt und trägt den Namen „Marvin“.

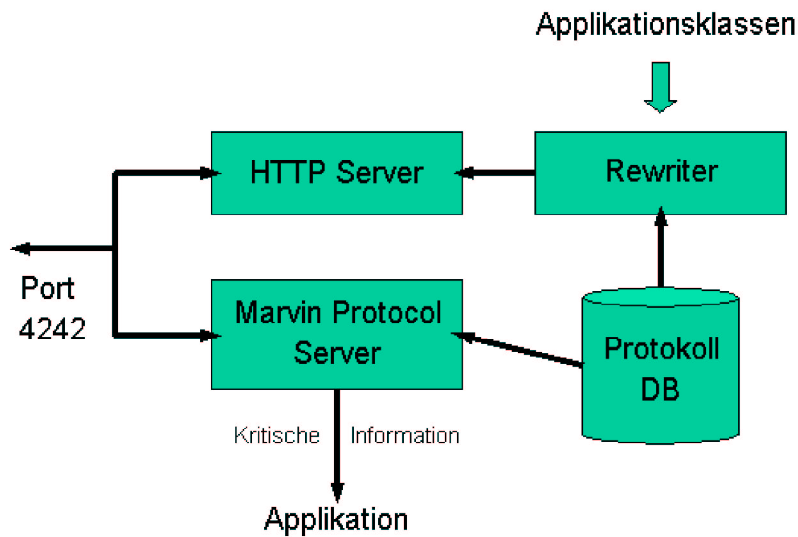
Basis der Umsetzung ist Java, womit sowohl die Programmiersprache Java als auch die Java-Plattform (Stichworte Virtual Machine, Bytecode) gemeint ist. Bei dem mobile Code handelt es sich um ein (austauschbares) Java-Applet. Die Clientsoftware besteht dementsprechend typischerweise aus einem Java-fähigen Web-Browser.

Die Marvin-Serversoftware ist eine Java-Applikation. Dies ist keine Notwendigkeit, die sich aus der Verwendung von Applets ergibt, sondern spiegelt im wesentlichen die Vorliebe des Autors wider. Darüber hinaus gibt es aber auch gute Gründe, eine Applikation dieser Art in Java zu entwickeln; als Stichworte sind Netzwerkfähigkeiten, objektorientiertes Design, Plattformunabhängigkeit und Multithreading zu nennen.

Eine kurze Bemerkung zu den Programmbeispielen: Die Beispielklassen zur Veranschaulichung der Obfuskation werden, obwohl die Obfuskation auf Bytecode-Ebene operiert, in dieser Arbeit nicht als Bytecode, sondern im Quelltext abgedruckt, da dies kürzer und wesentlich leichter verständlich ist. Zur Rückumwandlung des obfuszierten Bytecodes in Java-Quellcode wurde der für nichtkommerzielle Zwecke frei verfügbare Dekompiler Jad [JAD] verwendet, der nach Erfahrung des Autors das ausgereifteste Tool dieser Art darstellt.

3.1 Komponenten

Das folgende Schaubild stellt sehr grob die Struktur von Marvin dar:



Marvin ist über einen Port (standardmässig 4242) von aussen erreichbar. Der HTTP Server liefert Programmcode an den Browser, der Marvin Protocol Server kommuniziert direkt mit dem Applet. Beide Server hören auf dem gleichen Port; anhand der ersten Bytes einer eingehenden Verbindungen wird erkannt, welcher Server zuständig ist.

Die Protokoll-Datenbank generiert Protokollvariationen und hält diese zum Abruf bereit. Die Protokollbeschreibung wird zunächst vom Rewriter benötigt, der sie in den Programmcode einbettet. Später wird sie vom Marvin Protocol Server verwendet, um die Integrität des Clients zu prüfen und die kritische Information zu dekodieren. Diese wird schließlich an die jeweilige Anwendung übergeben, wo diese z.B. in einer Datenbank abgelegt wird.

3.2 Der Rewriter

Der Rewriter ist die Programmkomponente, die das Obfuszieren der Java-Klassen durchführt. Er lädt jeweils eine Klasse, transformiert sie und speichert sie. Auch der Bytecode innerhalb einer Methode wird quasi linear verarbeitet – Befehl für Befehl wird gelesen und sofort ausgegeben.

Durch diese direkte Verarbeitung werden wenig temporäre Datenstrukturen angelegt, was der Effizienz zugute kommt. Nachteilig ist, dass sich komplexe Transformationen schwerer implementieren lassen. Bei einigen der in Kapitel 4 vorgestellten Weiterentwicklungen von Marvin wurde dies schmerzhaft deutlich – sie wären in einer auf symbolischer Verarbeitung basierenden Architektur einfacher zu realisieren gewesen.

Für das eigentliche Lesen und Schreiben der .class-Dateien, das recht mühsam zu implementieren ist, habe ich bestehende Hilfsklassen aus Open-Source-Projekten eingesetzt.

Um ein wenig zu verdeutlichen, wie das Rewriting vor sich geht, werde ich die Struktur der zentralen Methode *processClass* der Klasse *Rewriter* skizzieren. Die Methode obfusziert genau eine Klasse; das Ergebnis wird als Byte-Array im .class-Format geliefert und kann direkt als .class-File auf die Platte gespeichert oder einem .jar-Archiv hinzugefügt werden.

```
byte[] processClass(Klass k) throws RewritingException {
  [...]
  // Schleife über alle Methoden
  for (int i = 0; i < methods.length; i++) {
    [...]
    // Schleife über den Rumpf der Methode (Bytecode)
    while (p < codeLen) {
      // Nächsten Opcode laden
      int op = codestream.readUnsignedByte();
      switch (op) { // Verzweigung abhängig vom Opcode
        [Opcode-Behandlung]
      }
    }
    [...]
  }
  [...]
}
```

Der Teil, der mit „Opcode-Behandlung“ bezeichnet ist, betrachtet den aktuellen Opcode und gibt ihn verändert oder unverändert wieder aus. Nicht verändert werden beispielsweise parameterlose Opcodes (wie *dup*, *return*, *nop*). Alle Opcodes, die Klassen-, Methoden- oder Feldnamen referenzieren, müssen dagegen transformiert werden. Als Beispiel sei die Behandlung von *invokeinterface* (Aufruf einer in einem Interface deklarierten Funktion) gezeigt:

```
case op_invokeinterface: {
  // Parameter lesen

  t = codestream.readUnsignedShort();
  p += 2; // Zeiger in Codebuffer erhöhen

  // Zweiten Parameter ignorieren
  // (bezeichnet Anzahl der Argumente; wird ohnehin
  // erneut berechnet)

  codestream.readShort();
  p += 2; // Zeiger in Codebuffer erhöhen

  // Methodenreferenz aus Constant Pool laden

  CONSTANT_InterfaceMethodref_info info =
   (CONSTANT_InterfaceMethodref_info) k.getConstant(t);

  // Befehl schreiben; dabei Interfacename, Methodenname,
  // Argumenttypen und Rückgabetyp transformieren

  w.add((byte) op, transformClassName(cInfo,
    info.getClassName()),
    transformMethodName(cInfo,
                        info.getClassName(),
                        info.getName(),
                        info.getSignature()),
    transformTypeString(cInfo,
                        info.getParameters()),
    transformTypeString(cInfo,
                        info.getReturnType()));

  // Aus switch-Statement herausspringen
  break;
}
```

3.3 Protokoll

Das von Marvin unterstützte Client-Server-Protokoll ist recht simpel gestrickt. Den Verbindungsaufbau nimmt das Applet beim Start vor. (Die genaue Programmstelle, an der dies geschieht, kann beim Rewriting zufällig variiert werden, um die Identifikation der für die Kommunikation zuständigen Klassen zu erschweren.)

Das Applet öffnet eine Socket-Verbindung zum Server. Diese Verbindung muss über die gesamte Laufzeit des Applets bestehen bleiben; der Aufbau einer weiteren Verbindung wird vom Server zurückgewiesen. Dies ist ein wichtiges Sicherheitselement: eine bestehende Verbindung von aussen zu manipulieren ist sehr viel schwieriger (im nachhinein sogar so gut wie unmöglich), als von einem

anderen Prozess aus eine neue Verbindung aufzubauen und hierüber manipulierte Daten zu übertragen.
Einmal pro Minute sendet das Applet eine „Idle“-Nachricht, um sicher zu stellen, dass die Verbindung nicht wegen Inaktivität abgebaut wird (manche ISDN-Verbindungen beispielsweise werden so konfiguriert, dass sie nach einer gewissen Zeitspanne ohne Netzaktivität die Leitung freigeben – und dadurch bestehende TCP-Verbindungen kappen).
Über die Verbindung werden folgende Arten von Informationen ausgetauscht:

3.3.1 Ereignisse

Eine Auswahl von im vorherein festgelegten Codes, die keine weiteren Parameter benötigen. Für jedes dieser Ereignisse wird bei der Protokolldefinition eine 32-Bit-Zahl zufällig festgesetzt.
Ereignisse werden grundsätzlich vom Applet zum Server übertragen.
Marvin selbst definiert nur ein einziges Ereignis, das Startereignis. Es wird ganz zu Beginn der Kommunikation übertragen, um eine erste Authentifikation zu leisten.
Weitere Ereignisse können (und sollten) von der Anwendung definiert werden. Zum einen dient dies der Sicherheit des Protokolls; zum anderen können Ereignisse auch genutzt werden, um Informationen zu übertragen, die die *kritische Information* ergänzen oder bestätigen.

3.3.2 Parametrische Ereignisse

Ein Ereignis, das von einem oder mehreren Parametern gefolgt wird. Direkt unterstützt werden von Marvin Ereignisse mit einer festen Anzahl von 32-Bit-Integern als Parametern. Für komplexere Parameter, etwa Strings, muss zur Zeit sowohl für die Client- wie für die Serverseite ein wenig anwendungsspezifischer Code geschrieben werden.
Die kritische Information wird im allgemeinen als ein oder mehrere parametrische Ereignisse übertragen.

3.3.3 Server-Antworten

Für den „Rückkanal“, also die Datenübertragung vom Server zum Applet, ist eine Funktion vorgesehen, die einen beliebigen String überträgt. Dieser wird automatisch kodiert und dekodiert.
Der String kann die Antwort auf ein Ereignis sein, aber auch auf eigene Veranlassung des Servers gesendet werden. In jedem Fall muss das Applet den String aber explizit entgegennehmen. Und selbstverständlich muss aufgrund stillschweigender Vereinbarungen klar sein, wie der Inhalt zu interpretieren ist.
In bestehenden Anwendungen wird diese Funktion eingesetzt, um den Inhalt von Konfigurationsdateien herunterzuladen. Vorher passierte dies über einen simplen HTTP-Abruf, was eine grosse Sicherheitslücke darstellte (die in einem Fall tatsächlich ausgenutzt wurde).

3.3.4 Variationen des Protokolls

Wie in Abschnitt 2.2 erläutert, ist einer der Grundpfeiler des von Marvin geleisteten Schutzes die zufallsbasierte Abwandlung des Client-Server-Protokolls. Folgende Variationsmöglichkeiten sind momentan vorgesehen:

- Die numerischen IDs aller Ereignisse werden zufällig festgelegt.
- Für Parameter von parametrischen Ereignissen wird ein Pseudo-Ereignis definiert; mit der ID dieses Ereignisses wird der Parameterwert vor der Übertragung moduliert.
- Ereignisse können optional sein, das heisst, sie tauchen nur in einigen Inkarnationen des Protokolls auf.

Diese im Grunde simplen Variationen genügen, um ein wesentliches Ziel zu erreichen: Das Protokoll für eine konkrete Applet-Auslieferung lässt sich nicht „erraten“, sondern ist im ausgelieferten Programmcode verborgen – im optimalen Fall sehr versteckt und auf viele Programmzeilen verteilt.

3.4 API

Marvin bietet eine spezielle API (Application Program Interface), die dazu dient, Einfluss auf die Applet-Server-Kommunikation zu nehmen. Ein Teil der Funktionen kann innerhalb des Applets aufgerufen werden; ein anderer Teil ist für serverseitig eingebundenen Code gedacht; und einige Funktionen lassen sich auf beiden Seiten der Verbindung einsetzen.

3.4.1 drjava.marvin.api.Talkback

Appletseitig wird primär die Klasse *Talkback* mit folgender Schnittstelle eingesetzt:

```
public class drjava.marvin.api.Talkback {
  public static void initTalkback();
  public static void send(int eventId);
  public static String slurp();
  public static void setIdleTask(Runnable task,
    int frequencyInMinutes);
}
```

Beschreibung der einzelnen Methoden:

- **initTalkback:** baut die Verbindung zum Server auf. Muss einmalig aufgerufen werden (möglichst direkt nach Initialisierung des Applets). Das Auslösen von Ereignissen ist erst nach Aufruf dieser Methode möglich.
- **send:** meldet ein einzelnes Ereignis an den Server. Die Ereignis-ID ist bereits die für das aktuelle Protokoll kodierte. Um diese zu ermitteln, wird im Normalfall vorher die Methode Hook.eventId(String name) aufgerufen (siehe nächsten Abschnitt).
- **slurp:** Der klangvolle Name dieser Methode („Schlürfen") bedarf wohl einer Erklärung: Die Methode dient dazu, eine Zeichenkette vom Server abzurufen. Dies ist nur möglich, wenn auf dem Server anwendungsspezifischer Code eingebunden ist, der eine Zeichenkette generiert. Der Zeichenketten-Übertragung kann das Senden eines Ereignisses voran gehen, mit dem der Client gezielt bestimmte Daten anfordert.
- **setIdleTask:** sorgt dafür, dass eine bestimmte Aufgabe im angegebenen Rhythmus wiederholt ausgeführt wird. Dies ist keine Funktionalität, die direkt

mit Marvin in Beziehung steht; da Marvin aber ohnehin einen Idle-Thread benötigt, hilft es Ressourcen sparen, diesen Thread auch für weitere Hintergrundtätigkeiten zu verwenden.

3.4.2 drjava.marvin.api.Hook

3.4.2.1 Schnittstelle

```
public class drjava.marvin.api.Hook {
  ////// Condition hooks

  public static boolean marvinized();

  public static boolean sometimes(String id);
  public static boolean maybeHere(String id, int num,
    int total);

  public static int authKey();

  ////// Value hooks

  public static String getHost();
  public static int getPort();

  public static int eventId(String name);
}
```

3.4.2.2 Beschreibung

Die Methoden der Klasse *Hook* sind keine gewöhnlichen Java-Methoden. Sie werden zwar ganz normal deklariert und aufgerufen. Der Methodenaufruf findet aber nicht zur Laufzeit statt; statt dessen simuliert der Rewriter den Aufruf und setzt das Ergebnis direkt an der Aufrufstelle ein. Hooks sind also die Stellen, in denen sich der Rewriter unmittelbar in den Programmcode „einhakt" (daher der Name).

Nehmen wir beispielsweise an, im Programmtext stehe folgende Zeile:

```
// Original-Code
Talkback.send(Hook.eventId(„Key pressed"));
```

Hier wird die ID des Ereignisses „Key pressed" ermittelt und an den Server gesendet. Der Rewriter erzeugt daraus etwa dies:

```
// Transformierter Code
Talkback.send(195872643);
```

Der Klartext-Name des Ereignisses taucht im endgültigen Bytecode also nicht mehr auf.

3.4.2.3 Methoden im Detail

Die Methoden der Klasse Hook teilen sich in so genannte *condition hooks* und *value hooks*. Der Unterschied liegt im Rückgabetyp: condition hooks geben einen boolschen Wert zurück, value hooks dagegen int oder String. Ausserdem werden die beiden Typen vom Rewriter anders behandelt. Bei value hooks wird der zurückgegebene Wert als Konstante in den Bytecode-Strom eingefügt. Condition hooks werden dagegen primär in if-Abfragen eingesetzt und führen dazu, dass einer der beiden Zweige der Abfrage bei der Obfuskation komplett wegfällt. Zunächst zu den *condition hooks*:

- **marvinized:** liefert true, wenn das aktuelle Applet von Marvin bearbeitet und ausgeliefert wurde. Diese Funktion ist nützlich, um ein Applet ggf. auch ohne Marvin verwenden zu können, ohne zwei Versionen der Codebasis erstellen zu müssen.
  Ein Beispiel:

```
if (Hook.marvinized())
  configuration = Talkback.slurp();
else
  configuration = loadURL(„http://server/config.txt“);
```

- **sometimes:** gibt mit einer Wahrscheinlichkeit von 50% *wahr* zurück. Die ID dient dabei zur Identifikation einer bestimmten Option; wird *sometimes* ein zweites Mal mit der gleichen ID aufgerufen, ist das Ergebnis garantiert identisch. (Dies gilt selbstverständlich nur innerhalb einer Applet-Auslieferung.) Ausserdem kann der serverseitige Anwendungscode anhand der ID abfragen, ob ein bestimmter Programmteil im Applet enthalten ist oder nicht; dies kann er u.U. zur semantischen Überprüfung der Applet-Server-Kommunikation nutzen.
  Als Beispiel sei ein optionales Ereignis gezeigt, das die Ausführung eines bestimmten Programmschritts meldet:

```
if (Hook.sometimes(Hook.eventId(„Step 3?“)))
  Talkback.send(Hook.eventId(„Step 3“));
performStep3();
```

- **maybeHere:** dient zur Auswahl eines Programmzweigs aus einer Gesamtzahl von *total* Zweigen. Die ID dient der Identifikation zusammen gehöriger Auswahlzweige.
  Beispiel: Die Talkback-Initialisierung wird an einem von drei Programmpunkten ausgeführt.

```
if (Hook.maybeHere("initTalkback", 1, 3)) {
  Talkback.initTalkback();
}

[...do some initialization...]

if (Hook.maybeHere("initTalkback", 2, 3)) {
  Talkback.initTalkback();
}

[...do some more initialization...]

if (Hook.maybeHere("initTalkback", 3, 3)) {
  Talkback.initTalkback();
}
```

- **authKey:** wird von Marvin intern verwendet (liefert eine ID, die die Appletversion gegenüber dem Server identifiziert).

Die Liste der *value hooks* ist kürzer als die der condition hooks und schnell beschrieben:

- **getHost:** liefert den Namen des Servers, von dem das Applet geladen wurde
- **getPort:** liefert die Portnummer, von der das Applet geladen wurde
- **eventId:** liefert die ID zur Klartext-Bezeichnung eines Events

3.4.2.4 Verwendung im Kontext

Hooks sind das primäre Mittel, um das Applet-Server-Protokoll möglichst versteckt und verteilt in den Programmtext des Applets einzubetten.
Durch die Ersetzung der Hook-Methodenaufrufe zur Obfuskationszeit lässt sich das Protokoll nur mit großem manuellem Aufwand oder der Entwicklung komplexer Analyseprogramme aus dem Bytecode extrahieren. Um beispielsweise überhaupt festzustellen, wie viele Zweige eine *maybeHere*-Auswahl besitzt, muss der Angreifer eine große Anzahl von Appletversionen herunterladen und analysieren.

4 Verbesserungen der Implementation

4.1 Stringliterale

4.1.1 Einführung

Java-Programme enthalten i.a. viele Stringliterale, d.H. fest kodierte Zeichenketten. In diesem stilisierten Beispiel:

```
class StringLiterals {
  void sendData(String data) {
    Socket socket =
      new Socket("www.secretserver.com", 123);
    PrintWriter w =
      new PrintWriter(socket.getOutputStream());
    w.println("Secret data: "+data);
    w.close();
  }
}
```

sind zwei solche Literale enthalten (zur Kenntlichmachung unterstrichen).

4.1.2 Aktuelle Situation

Stringliterale werden beim Obfuscating nicht verändert. Die Beispielklasse sieht obfusziert etwa so aus:

```
//Decompiled by Jad v1.5.7f. (C) 2000 Pavel Kouznetsov.

class _xQ
{
  void _(String s)
  {
    Socket socket =
      new Socket("www.secretserver.com", 123);
    PrintWriter printwriter =
      new PrintWriter(socket.getOutputStream());
    printwriter.println("Secret data: " + s);
    printwriter.close();
  }
}
```

Man sieht: die einzigen Bezeichner, die erhalten blieben, sind die Namen von Systemklassen und -methoden – und die Stringliterale.

Stringliterale können das Wiederfinden von obfuszierten Klassen u. U. drastisch erleichtern. Ein Befehl in der Art von

```
grep -c secretserver *.class
```

würde genügen, um die obige Klasse in einem Archiv mit womöglich hunderten von Klassen ausfindig zu machen – auch wenn sie nicht mehr „StringLiterals“ heisst, sondern „_xQ“.

4.1.3 Verbesserungsvorschlag

Alle Stringkonstanten werden kodiert gespeichert. Um Abhängigkeiten zwischen Klassen zu vermeiden und die Analyse zu verkomplizieren, wird in jeder Klasse, die Stringliterale enthält, eine Dekodierfunktion angelegt. Der Kodieralgorithmus sollte Parameter besitzen, die per Zufall variiert werden.
Der Aufruf der Dekodierfunktion kann zu verschiedenen Zeitpunkten erfolgen:

1. beim Laden der Klasse (alle Strings werden dekodiert und in statischen Variablen gespeichert)
2. bei jedem Aufruf eines Stringliterals

Beide Varianten sind in etwa gleich aufwändig zu realisieren. Der klare Vorteil der ersten Variante ist aber, dass zur Laufzeit kein Performanceverlust entsteht (ausser beim Programmstart).
Was für eine Art von Kodierung ist sinnvoll? Diese Frage kann am besten im Hinblick auf das Reverse Engineering beantwortet werden. Wie gelangt ein Angreifer an die dekodierten Strings? Entweder durch Ausführen der Dekodierfunktion(en) oder durch Analyse derselben. Der Algorithmus sollte also zumindest so beschaffen sein, dass eine Analyse der Dekodierfunktion nicht mit realistischem Aufwand möglich ist.
Aufbauend auf die Grundfunktionalität der String-Kodierung sind weitere „Vernebelungs“-Massnahmen denkbar:

- Verstecken der Dekodierfunktion: Variation des Namens, der Signatur (Hinzufügen weiterer Parameter); Generierung von ähnlich aussehenden, aber nicht verwendeten Funktionen

- Strings können u.U. anhand ihrer Länge gefunden werden. Dies kann verhindert werden, indem man alle (kodierten) Strings in einen einzigen zusammenfasst und diese durch Angabe von Offsets und einer kodierten Längenangabe extrahiert. Wenn Codegröße nicht die höchste Priorität ist, kann dieser „Stringblock“ künstlich vergrössert werden, um die Identifikation von Klassen anhand der Länge dieses Blocks zu verhindern.

4.1.4 Umsetzung

Die grundlegende Transformation besteht darin, im Bytecode Ladebefehle Stringkonstanten im Bytecode durch Zugriffe auf statische Felder zu ersetzen. Aus

```
ldc „www.secretserver.com"
```

wird beispielsweise

```
getstatic _ Ljava/lang/String;
```

, also das Laden des statischen Felds mit dem Namen „_“, das vom Typ java.lang.String ist.

Diese neu eingeführten statischen Variablen werden mit den kodierten Strings vorbelegt und beim Initialisieren der Klasse dekodiert.

Ein weiterer Fall, der zu beachten ist, sind bereits vorhandene statische Variablen, die mit einem Stringwert initialisiert werden. Hier wird im Prinzip genauso verfahren wie im Fall der im Bytecode eingebetteten Stringliterale; nur dass es nicht nötig ist, eine neue Variable anzulegen.

Der verwendete Kodieralgorithmus ist parametrisch, aber in seiner Struktur fest. Der Parameter besteht aus einem Integer, wird pro Klasse festgelegt und in die Dekodierfunktion eingebettet.

Wird Stringverschlüsselung aktiviert, führt Obfuskation und Dekompilation der Beispielklasse zu folgendem Ergebnis (leicht gekürzt):

```
//Decompiled by Jad v1.5.7f. (C) 2000 Pavel Kouznetsov.

class StringLiterals {
  void _(String s) {
    Socket socket = new Socket(_, 123);
    PrintWriter printwriter =
    new PrintWriter(socket.getOutputStream());
    printwriter.println(a + s);
    printwriter.close();
  }

  private static String _(String s) {
    int i = s.length();
    char ac[] = new char[i];
    for(int j = 0; j < i; j++)
      ac[j] = (char)(s.charAt(j) ^ 0xfffe9df2);

    return new String(ac);
  }

  private static String _ =
"\u9D85\u9D85\u9D85\u9DDC\u9D81\u9D97\u9D91\u9D80\u9D97\u9D
86\u9D81\u9D97\u9D80\u9D84\u9D97\u9D80\u9DDC\u9D91\u9D9D\u9
D9F";
  private static String a =
"\u9DA1\u9D97\u9D91\u9D80\u9D97\u9D86\u9DD2\u9D96\u9D93\u9D
86\u9D93\u9DC8\u9DD2";

  static {
    _ = _(_);
    a = _(a);
  }
}
```

Es fällt auf, dass die Literale tatsächlich durch Ketten auf den ersten Blick nicht entzifferbarer Hexadezimal-Codes ersetzt wurden. (Die kodierten Strings nutzen den vollen 16-bittigen Definitionsbereich des Unicode-Zeichensatzes aus; daher werden sie in Quelltexten hexadezimal kodiert.)

Beim zweiten Hinsehen erkennt man allerdings auch, dass die Kodierung nicht sehr komplex ist. Zum Beispiel spielt die Position eines Zeichens keine Rolle; gleiche Zeichen werden also immer wieder in gleiche Zeichen abgebildet. Auch ist die Wiederholung des höherwertigen Bytes 9D recht unschön; diese ist ein Artefakt der simplen XOR-Kodierung (in den meisten Unicode-Texten ist das höherwertige Byte über weite Strecken konstant). Hier könnte schon eine leichte Abwandlung des Kodieralgorithmus eine deutliche Verbesserung bringen.

4.1.5 Bewertung

Man kann von einem ausreichenden Erfolg sprechen. Führen wir uns das ursprüngliche Ziel noch einmal vor Augen: Es sollte die Identifikation einzelner Klassen durch eine einfache Volltextsuche über den .class-Dateien verhindert werden. Dies wurde definitiv erreicht. Da jede Klasse einen anderen Kodierschlüssel verwendet, ist es ebenfalls sinnlos, den gesuchten Text manuell zu kodieren und nach der kodierten Form zu suchen.

4.2 Dynamische Kodierung der kritischen Information

4.2.1 Aktuelle Situation

Von den denkbaren Angriffsstrategien führt momentan eine wohl am ehesten zum Erfolg: der Einsatz eines speziellen Proxies.
Es handelt sich im Grunde um eine Variante des aus der IT-Sicherheit bekannten „Man in the middle"-Angriffs: Man schaltet eine eigene Software zwischen Client und Server, die jedem Kommunikationspartner jeweils vortäuscht, die andere Seite zu sein. Wenn der Angreifer das Protokoll kennt oder ermitteln kann, kann er Datenpakete gezielt verändern, ohne dass dies einer der Parteien auffällt.
Auf Marvin-Applets lässt sich dieser Angriff technisch in zwei Varianten durchführen:

1. Entwicklung einer Proxy-Applikation auf Netzwerkebene
2. Veränderung der Java-Laufzeitklassen (java.net.*) des verwendeten Browsers

Der Effekt ist in beiden Fällen im wesentlichen identisch.
Man wird den Proxy so einstellen, dass er nach dem Verbindungsaufbau seitens des Applets selbst eine Verbindung zum Server öffnet und zunächst alle Daten (in beide Richtungen) unverändert weitergibt.
Dann beginnt man mit der Analyse des Applets (der Bytecode muss beim Herunterladen zusätzlich gespeichert worden sein, z.B. ebenfalls über einen

Proxy) mit dem Ziel, die Kodierung der kritischen Information zu ermitteln.
Gelingt dies, ist der Angriff geglückt: Der Proxy greift während der Übermittlung der kritischen Information ein und überträgt statt dieser eine manipulierte Variante.
Durch das Aufrechterhalten der Verbindung umgeht man also die Zeitbeschränkung t_{max} für die Rückmeldung des Applets.

4.2.2 Verbesserungsvorschlag

Die Idee für eine Erschwerung des Angriffs besteht darin, für die Kodierung der gefälschten kritischen Information eine Zeitbeschränkung t_{crit} einzuführen. Dies geschieht, indem in die Kodierung Parameter einbezogen werden, die erst kurz vorher geliefert werden. Ich bezeichne dies als *dynamische Kodierung der kritischen Information*.
t_{crit} kann sehr kurz gewählt werden, viel kürzer als t_{max}. Es genügt, wenn folgendes gegeben ist:

$$\begin{aligned} t_{crit} >= \; & Latenz_{Server->Client} + Übertragungszeit_{Parameter} + \\ & Rechenzeit_{Kodierung} + \\ & Latenz_{Client->Server} + Übertragungszeit_{Kritische\ Information} + \\ & Toleranz \end{aligned}$$

Ein realistischer Wert wäre z.B.

$$\begin{aligned} t_{crit} = \; & 3s + 0,5s + \\ & 2s + \\ & 3s + 0,5s + \\ & 1s \\ & = \mathbf{10s} \end{aligned}$$

(Der Rechenaufwand für die Kodierung wird im Millisekundenbereich liegen; in den 2 Sekunden sind Dinge wie Hintergrundprozesse und Swapping eingerechnet.)
Eine manuelle Kodierung der kritischen Information müsste also in 10 Sekunden (abzüglich der Übertragung) erfolgen. Dies ist schon bei einem einfachen Kodieralgorithmus nicht mehr machbar.

Die Wirkung der dynamischen Kodierung der kritischen Information besteht also darin, den Aufwand des Angriffs zu erhöhen: nun ist während des Ablaufs des anvisierten Applets noch eine Programmierleistung des Angreifers erforderlich. Voraussetzung für die Effektivität der Maßnahme ist dabei, dass die Struktur des Algorithmus zwischen den Appletversionen variiert wird.

4.2.3 Umsetzung

4.2.3.1 API-Erweiterungen

Für die dynamische Übermittlung der kritischen Information wurden folgende Klassen und Methoden eingeführt:

- Appletseitig die Klasse *CriticalInformationSender* mit der statischen Methode
  ```
  void send(int twoColonsEventId, int data);
  ```
  Diese erhält eine Event-ID und die eigentlichen Daten. Sie sendet die Event-ID, die kodierten Daten und eine Prüfsumme an den Server. Das Präfix „twoColons“ deutet an, dass es sich um ein Ereignis mit zwei Parametern handeln muss, also zwei Doppelpunkten im Namen.
  Ein Beispiel zur Verwendung:
  ```
  CriticalInformationSender.send(
    Hook.eventId(„wichtigerWert::“, wichtigerWert);
  ```
- Serverseitig steht in der API-Klasse *SessionLink* die Methode
  ```
  public int decodeCriticalData(int[] args);
  ```
  zum Dekodieren und Prüfen eines empfangenen der kritischen Daten zur Verfügung.
  Die Zeitbeschränkung t_{crit} kann in der Projekt-Konfigurationsdatei oder in der Projektklasse durch Überschreiben der Methode
  ```
  public int getCriticalTimeout();
  ```
  verändert werden. Als Vorgabe sind 10 Sekunden eingestellt.

4.2.3.2 Kodierung

Der Kodieralgorithmus besitzt drei Parameter:

- critical_p1
- critical_p2
- critical_xor

p1 und *p2* gehen in die Berechnung der Prüfsumme ein; *xor* wird zur Kodierung der Daten verwendet.
Der Algorithmus sieht folgendermaßen aus:

```
// data ist Eingabewert

int encoded = data^critical_xor;
int checksum = 0;
int p = critical_p1;
for (int i = 0; i < critical_p1 & 255; i++)
  checksum = (checksum^(p += critical_p2))+i;

// Zum Server übertragen: Paar (encoded, checksum)
```

4.2.3.3 Protokoll

p2 und *xor* werden über den Hook-Mechanismus direkt in den Programmcode eingebettet. *p1* wird dagegen erst unmittelbar vor der Übertragung der kritischen Information an das Applet gesendet und markiert den Beginn des durch t_{crit} beschränkten Zeitintervalls.
Der Ablauf bei der Übertragung der kritischen Information ist folgender:

1. [Client->Server] Ereignis $get_critical_p1
   (Anfordern des Parameters *p1*)
2. [Server->Client] Wert von *p1* als String
3. [Client->Server] Ereignis mit zwei Parametern:
   1. xor(*data*, *critical_xor*)
   2. Prüfsumme

4.2.4 Bewertung

Das grundlegende Ziel wurde erreicht: Das Kodieren einer fingierten kritischen Information lässt sich nicht mehr allein nach Studium des dekodierten Applet-Codes durchführen. Es ist vielmehr zusätzlich erforderlich, innerhalb der bestehenden Verbindung Information vom Server anzufordern (*p1*) und darauf in sehr kurzer Zeit einen Kodieralgorithmus anzuwenden.
Zu verbessern bleiben zwei Dinge:

- Der Kodieralgorithmus ist zwar parametrisch, aber in seiner Struktur fest. Es genügt für einen Angreifer also, diesen einmalig nachzuprogrammieren. Über *condition hooks* wäre es leicht möglich, die Struktur des Algorithmus bei jeder Auslieferung deutlich zu variieren.
- Es ist ungünstig, dass das Kodieren und Versenden in einer separaten Klasse passiert. Diese ist aufgrund ihrer geringen Grösse relativ leicht zu identifizieren. Besser wäre die Einbettung der Methoden an der Aufrufstelle (sog. „inlining“). Dies ist allerdings mit der aktuellen Architektur des Rewriters schwer zu realisieren.

4.3 Verstecken von Referenzen auf Systemklassen

4.3.1 Einführung

Obfuskation kann die Struktur und Funktionsweise von anwendungsspezifischen Klassen sehr weitgehend verschleiern. In vielen Fällen können beispielsweise sämtliche Bezeichner durch nichts sagende Zeichenkombinationen ersetzt werden. Jede sinnvolle Anwendung verwendet allerdings an einer oder mehreren Stellen Systemklassen und/oder -methoden, also Schnittstellen zu Betriebssystem, Interpreter oder Virtual Machine. Die Programmstellen, an denen dies geschieht, sind oft ein guter Ausgangspunkt für das Reverse Engineering; bestimmte Symbole müssen hier zwangsläufig auftauchen, z.B. Namen von Systemmethoden.

Abhängig von der verwendeten Programmiersprache und Systemumgebung gibt es verschiedene Möglichkeiten, Systemreferenzen zu verschleiern:

- In einer sehr hardwarenahen Umgebung kann oft auf den Einsatz von Systemfunktionen gänzlich verzichtet werden. Peripheriegeräte lassen sich hier auch über direkte Port- oder Speicherzugriffe ansprechen. Oft ist dies schon aus Performancegründen der bevorzugte Weg.
  Sind Anwendungen dieser Art in Maschinensprache geschrieben oder wurden sie aus einer Hochsprache in Maschinencode übersetzt, wird das Herausfiltern

dieser speziellen Speicherzugriffe aus den unzähligen, die auf Anwendungsdaten operieren, bereits sehr schwierig.
In Maschinensprache steht zudem noch eine weitere sehr mächtige Obfuskationstechnik zur Verfügung: Selbstmodifizierender Code. Hier fungieren bestimmte Teile des Programms als Code-Generatoren, die bestimmte andere Programmteile erst zur Laufzeit generieren. Die Flexibilität dieser Technik ist enorm; schon zu den Zeiten, als Obfuskation als Basis für Kopierschutz diente, wurde sie viel eingesetzt.

- Moderne Betriebssysteme verwalten zentral alle Hardwareressourcen. Anwendungen haben keinen unmittelbaren Zugriff auf die Hardware (dies wird durch Sicherheitsmechanismen im Prozessor sicher gestellt und ist, von Bugs in Betriebssystem oder Prozessor abgesehen, tatsächlich eine unüberwindliche Schranke). Alle Hardwarezugriffe müssen also über Betriebsystemfunktionen erfolgen.
  Ausserdem definieren Betriebssysteme dieser Art ein festes Dateiformat für Anwendungscode; Referenzen auf Systemfunktionen tauchen darin explizit auf und lassen sich leicht zurückverfolgen.
  Die Möglichkeiten zum Verstecken von Systemreferenzen sind dadurch gegenüber dem ersten Fall stark eingeschränkt. Selbstmodifizierender Code ist allerdings nach wie vor erlaubt. Eine andere Verschleierungsmöglichkeit bieten Systemroutinen, die den Zugriff auf Bibliotheken erlauben, deren Name erst zur Laufzeit angegeben werden muss.
- In einer Sprache wie Java, deren Kompilat im Normalfall nicht als Maschinencode, sondern als Bytecode vorliegt, sind die Einschränkungen noch grösser: Der einzige Weg, auf Systemressourcen zuzugreifen, führt über die von der Virtual Machine bereitgestellten Java-Laufzeitklassen. Das Format der Java-Classfiles ist zudem so angelegt, dass darin die Namen aller referenzierten Klassen im Klartext auftauchen.

Um Systemreferenzen in Java-Programmen zu verstecken, bieten sich im wesentlichen zwei Wege an:

1. Selbstmodifizierenden Code im wörtlichen Sinn kennt Java nicht – Klassen, die bereits in Verwendung sind, können nicht nachträglich modifiziert werden. Java bietet aber die Möglichkeit, neue Klassen zur Laufzeit zu definieren und

zu laden. Dies ermöglicht effektive, aber auch sehr aufwändige Ansätze zur fortgeschrittenen Obfuskation.

2. Seit der Version 1.1 bietet Java ein Feature namens *Reflection*. Reflection ermöglicht es, zur Laufzeit die Struktur von Klassen abzufragen (z.B. Methoden, Felder), Objekte einer unbekannten Klasse anzulegen und Methoden dynamisch aufzurufen.
   Klassen- und Methodennamen werden beim Einsatz von Reflection als Strings übergeben. Kombiniert man Reflection also mit der bereits implementierten String-Verschlüsselung, ist man im Besitz einer probaten Methode, Systemreferenzen zumindest vor einer oberflächlichen Suche zu verstecken.

Der zweite Ansatz stellt guten Erfolg in Aussicht und ist der bei weitem weniger aufwändige der beiden. Ich habe mich daher entschieden, diesen umzusetzen.

4.3.2 Aktuelle Situation

Referenzen auf Systemklassen und -methoden werden vom Marvin-Rewriter momentan überhaupt nicht verändert.

4.3.3 Verbesserungsvorschlag

Da Methodenaufrufe per Reflection um mindestens zwei Größenordnungen langsamer sind als direkte Aufrufe, sollten diese sparsam eingesetzt werden. Ein automatisches Verstecken aller Systemaufrufe scheidet daher aus. Ich habe mich dafür entschieden, das Verstecken als eine Option pro Systemklasse anzubieten. In der Konfigurationsdatei sieht die entsprechende Zeile dann z.B. so aus:

```
hideCalls=(„java.net.Socket“, „java.io.*“)
```

Es lassen sich sowohl Aufrufe statischer wie nicht-statischer (virtueller) Methoden behandeln. Bei den virtuellen Methoden wird die Wirkung allerdings dadurch geschmälert, dass der Typ des betrachteten Objekts nach wie vor direkt angegeben werden muss, beispielsweise bei der Deklaration von Variablen.

Dies liesse sich dadurch umgehen, dass Instanzen der betroffenen Klassen generell mit dem Typ *Object* übergeben und gespeichert werden. Dies wirft aber einige Folgeprobleme auf; teilweise wird es z.B. erforderlich sein, Casts von *Object* hin zu dem wirklichen Typ einzufügen, wenn externe Bibliotheken ein Objekt dieses

Typs erwarten. Diese weiter gehende Obfuskation werde ich daher vorerst nicht implementieren.

4.3.4 Umsetzung

Jede Klasse, die Methoden einer der ausgewählten Systemklassen aufruft, wird um eine oder mehrere *Proxy-Methoden* erweitert. Proxy-Methoden führen den eigentlichen Aufruf per Reflection aus und wandeln dabei Argumente und Rückgabewert um, wenn dies nötig ist (z.B. int<->Integer).
Pro *Signatur*, also pro Kombination aus Parametertypen und Rückgabetyp, wird eine Proxy-Methode erzeugt.
Die Einführung von Proxy-Methoden hat gegenüber der direkten Verwendung von Reflection an der eigentlichen Aufrufstelle mehrere Vorteile:

- Geringere Code-Grösse bei wiederholtem Aufruf (Reflection-Handling passiert zentral)
- Zugewinn an Obfuskation. An der Aufrufstelle werden nur die (verschlüsselten) Namen von Klasse und Methode geladen und die Proxy-Methode aufgerufen; dass Reflection im Spiel ist, wird an dieser Stelle nicht sichtbar. Da der Marvin-Rewriter zudem in der Lage ist, Methodennamen sehr aggressiv wieder zu verwenden, lässt sich oft auch nur nach Typanalyse der Parameter feststellen, ob hier eine Proxy- oder eine gewöhnliche Methode aufgerufen wird.

4.3.5 Beispiel

```
class CallHiding {
  public static void main(String[] args) {
    System.out.println(System.currentTimeMillis());
  }
}
```

Dieses Programm gibt die aktuelle Uhrzeit aus (in Sekunden seit 1970).
Obfuskation mit dem Parameter

```
hideCalls=("java.lang.System")
```

(encryptStrings wird automatisch aktiviert, wenn hideCalls angegeben ist) und anschliessende Dekompilation mit Jad liefert dieses Ergebnis:

```
class CallHiding {
    public static void main(String args[]) {
        System.out.println(_(_, a));
    }

    private static String a(String s) {
        int i = s.length();
        char ac[] = new char[i];
        for(int j = 0; j < i; j++)
            ac[j] = (char)(s.charAt(j) ^ 0x24a7);

        return new String(ac);
    }

    private static long _(String s, String s1){
        return ((Long)Class.forName(s).getMethod(s1, new
Class[0]).invoke(null, null)).longValue();
    }

    private static String _ =
"\u24CD\u24C6\u24D1\u24C6\u2489\u24CB\u24C6\u24C9\u24C0\u24
89\u24F4\u24DE\u24D4\u24D3\u24C2\u24CA";
    private static String a =
"\u24C4\u24D2\u24D5\u24D5\u24C2\u24C9\u24D3\u24F3\u24CE\u24
CA\u24C2\u24EA\u24CE\u24CB\u24CB\u24CE\u24D4";

    static {
        _ = a(_);
        a = a(a);
    }
}
```

Wie gewünscht ist nicht mehr ersichtlich, dass die Methode System.currentTimeMillis() aufgerufen wird. Das Auslesen des Felds System.out und der Aufruf von PrintStream.println() bleiben dagegen sichtbar. Zumindest die Referenz auf das Feld wäre analog zu dem Verstecken von Methodenaufrufen leicht zu eliminieren.

4.3.6 Bewertung

Vorbehaltlich der Schließung der verbleibenden Lücken (Zugriff auf Felder, virtuelle Methoden) kann man sagen, dass das Ziel, Referenzen auf Systemklassen zu verschleiern, von dem implementierten Verfahren erreicht wird.

4.4 Code Streaming

4.4.1 Aktuelle Situation

Die in 2.2 beschriebene Zeitbeschränkung t_{max} greift nur einmal pro Auslieferung eines Applets: zwischen dem Herausgeben des obfuszierten Programmcodes und der Rückmeldung von seiten des Applets. Danach gibt es, abgesehen von dem in 4.2 eingeführten t_{crit}, keine weiteren Zeitschranken.
Ein denkbares Angriffsszenario sieht daher so aus, das Applet normal zu starten, aber sich z.B. durch den Einsatz eines Proxies die Möglichkeit offen zu halten, im nachhinein in die bestehende Applet-Server-Verbindung einzugreifen.
Entscheidend ist hierbei, dass für die Analyse des Programmcodes die gesamte Laufzeit des Applets, also quasi beliebig Zeit zur Verfügung steht. Viele Applets lassen sich in ihrer Gesamt-Laufzeit nicht sinnvoll beschränken; dies gilt beispielsweise für Online-Spiele, die der Benutzer immer wieder spielen darf, ohne das Applet komplett neu laden zu müssen.
Dies liesse sich verhindern, wenn der Code des Applets während der Ausführung nicht statisch bliebe, sondern in einer Art „Stoffwechsel“ ständig ausgetauscht würde. Es müsste also ein dauerndes Nachladen von Programmcode passieren. Ich verwende dafür analog zum *Streaming* von Multimediadaten den Begriff *Code Streaming*.
Für jeden Codeblock wird wieder eine Zeitbeschränkung angesetzt.
Das Verfahren wird umso effektiver sein, je grösser der Prozentsatz an Programmcode ist, der in den Austausch einbezogen wird. Im Idealfall würde der gesamte Code des Applets regelmässig ausgewechselt.

4.4.2 Verbesserungsvorschlag

Viele der Applets, die von Marvin profitieren können, lassen sich in ihrem Ablauf in einzelne Phasen eingeteilen. Einige Beispiele:

- die einzelnen Runden, Levels oder auch ganzen Spieldurchgänge eines Online-Spiels
- einzeln bearbeitbare Datenpakete bei verteilten Berechnungen

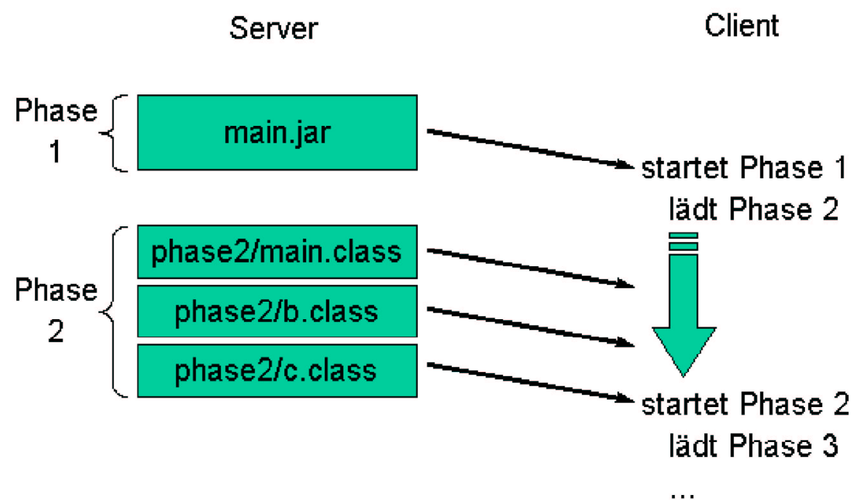
Die Abschnitte sollten jeweils mindestens ca. eine Minute lang sein.

Die Idee besteht darin, während einer Phase im Hintergrund den Programmcode für die nächste Phase zu laden. Da die Phasen einander strukturell sehr ähnlich sind, wird dies praktisch bedeuten, den gleichen Code noch einmal zu laden. Dieser wird aber einer erneuten Obfuskation unterzogen.
Dieses Verfahren erlaubt es, die einzelnen Abschnitte wie eigenständige Auslieferungen des Applets zu behandeln – es kann für die Dauer jedes Abschnitts eine Zeitbeschränkung t_p (p für *phase*) angesetzt werden.
Eine praktische Schwierigkeit liegt darin, dass unsignierte Java-Applets keine eigenen *ClassLoader* definieren dürfen; Code auf diese Art dynamisch auszuführen, ist also nicht möglich.
Eine Alternative wären signierte Applets. Diese erfordern allerdings immer die Bestätigung einer Sicherheitswarnung durch den Benutzer, die man oft gern vermeiden möchte.
Es gibt allerdings einen Trick, mit dem sich der gewünschte Effekt erreichen lässt. Hierzu wird das Verhalten des System-Classloaders für Applets ausgenutzt: Dieser fordert Klassen unter bestimmten Umständen erst dann vom Server an, wenn sie wirklich verwendet werden.

4.4.3 Umsetzung

Es gibt grundsätzlich zwei Möglichkeiten, Applet-Code zu übertragen: als einzelne .class-Dateien oder in einer einzigen Archivdatei (im Format .zip, .jar oder dem Microsoft-spezifischen .cab). Letzteres ist in praktisch jedem Fall vorzuziehen: Dank der Kompression ist die Gesamtzahl der zu übertragenden Bytes geringer, und vor allem ist nur eine einzige HTTP-Verbindung erforderlich. Der Nachteil des Klassenarchivs ist, dass alle Klassen am Stück übertragen werden. Sonst eher nebensächlich, wird dies in bezug auf das Code Streaming zum Problem; es geht ja gerade darum, den Programmcode etappenweise zu übertragen.
Die naheliegende Lösung besteht darin, alle Klassen als einzelne Dateien zu übertragen. Immerhin kann die erste Code Streaming-Phase zu einem Archiv zusammengefasst werden, so dass ein Teil des Effizienzverlusts ausgeglichen wird. Glücklicherweise ist dies auch der Zeitpunkt, an dem lange Ladezeiten am wenigsten tolerierbar sind – der Benutzer wartet auf den Aufbau des Applets und

kann noch nicht mit ihm interagieren. Das Laden der späteren Phasen geschieht parallel zum Ablauf des Applets und ist daher kaum zeitkritisch.
Das schematische Zeitdiagramm:



Jede Phase wird im wesentlichen wie eine eigene Applet-Auslieferung behandelt: Die Codebasis wird ausgetauscht, das Protokoll neu definiert und eine neue Verbindung eröffnet. Die aus der vorherigen Phase bestehende Verbindung wird abgebaut.

4.4.4 Bewertung

Code Streaming ist eine sehr mächtige Technik. Es hebt den Sicherheitsrückstand auf, den lang laufende Applets bisher hatten. Es ist jetzt nicht mehr möglich, die Zeitbeschränkung t_{max} zu umgehen – ein Angreifer muss Dekompilation, Analyse und Veränderung des Codes eines Abschnitts immer in einer Zeit von t_p bewältigen – danach ist seine Arbeit wertlos, da bereits ein neuer Abschnitt beginnt.
Die Kosten des Code Streamings sind demgegenüber eher gering:

- erhöhter Aufwand bei Anpassung des Applets (Übergabe von Daten an die nächste Phase und Aktivierung dieser)
- größere zu übertragende Datenmenge

- erhöhter Speicherbedarf; der Programmcode der abgeschlossenen Phasen kann aus technischen Gründen nicht „garbage collected“, also aus dem Hauptspeicher entfernt werden
- Wenn ein Angreifer den ausgelieferten Code über einen Proxy mitschneidet und abspeichert, kann er unauffällig „Material“ zur Analyse der Obfuscation-Mechanismen und Protokollvariationen sammeln. Ohne Code Streaming müsste er dafür das Applet sehr häufig aufrufen, was bei Vorhandensein eines serverseitigen Monitorings eventuell auffällt.

5 Gesamteinschätzung des verbesserten Systems

5.1 Bisherige Schwachpunkte

Die sicherheitsrelevanten Schwächen des Marvin-Systems lagen bisher in zwei Bereichen:

- **Qualität der Obfuskation.** Bei der Art und Weise, in der wiederholte Obfuskation hier als Mittel zum Schutz mobilen Codes eingesetzt wird, ist einer der wichtigsten Aspekte ist die Verhinderung der Identifikation von Klassen und Methoden. Diese Identifikation wurde bisher dadurch erleichtert, dass Stringliterale und Referenzen auf Systemklassen im Klartext erhalten blieben.

- **Rigorosität der Zeitbeschränkung.** Zwischen Auslieferung und Beginn der Ausführung des Applets besteht eine feste Zeitbeschränkung t_{max}; während der Ausführung erlauben aber bestimmte Tricks die Vornahme von Manipulationen ohne Zeitdruck (s. Abschnitt 4.4.1).

5.2 Erreichte Fortschritte

Die in dieser Arbeit entwickelten Verbesserungen betreffen beide im vorigen Abschnitt aufgeführten Punkte.

Die Kodierung von Stringliteralen (4.1) und das Verstecken von Referenzen auf Systemmethoden (4.3) erhöhen den Grad der Obfuskation. Die dynamische Kodierung der kritischen Information (4.2) lässt sich nicht exakt einordnen – sie trägt gewissermassen zur „Obfuskation“ des Protokolls bei.

Das Code Streaming (4.4) schließlich zielt auf den zweiten Problembereich; es garantiert eine ständige Zeitbeschränkung t_p. Diese wird in der Praxis oft höher

liegen als t_{max}, aber üblicherweise noch im einstelligen Minutenbereich, was immer noch eine sehr gute Sicherheit bietet.

5.3 Verbleibende Risiken

Die vorgenommenen Verbesserungen des Systems schließen alle wesentlichen Lücken. Unter der Voraussetzung, dass eine Anwendung bestimmte Bedingungen erfüllt – nicht zu geringe Codegröße, gut verteilte Einbettung des Protokolls – schützt Marvin diese nun sehr zuverlässig vor unerwünschten Eingriffen.
Erfolg versprechend ist nur noch einzige Angriffsstrategie: die Entwicklung eines Tools zum automatischen Reverse Engineering. Um besser darüber diskutieren zu können, geben wir diesem hypothetischen Tool einen Namen – nennen wir es den *Re-Rewriter*.
Der Re-Rewriter ist nicht einfach ein Dekompiler – er muss viel mehr leisten. Seine Aufgabe ist, den von Marvin ausgelieferten Code eines Applets zu analysieren, die für die betreffende Anwendung interessanten Programmstellen zu identifizieren und diese so zu verändern, dass die gewünschte Änderung der Programmfunktion erreicht wird. Dies alles muss im wesentlichen vollautomatisch geschehen, da für Eingriffe des Benutzers nur sehr wenig Zeit zur Verfügung steht.
Es ist denkbar, solch einen Re-Rewriter zu entwickeln. Einfach ist dies aber mit Sicherheit nicht – es verlangt sehr gute Kenntnisse des Java-Bytecodes und der Virtual Machine, flexible Libraries zur Analyse und Manipulation von Bytecode, eingehende Analyse der von Marvin eingesetzten Obfuscating-Varianten und die Implementation komplexer Heuristiken.
Wegen der ständigen Neu-Obfuskation ist es auch nicht möglich, ein einmalig „gecracktes“ Applet beispielsweise im Internet zu verteilen und damit anderen, die keine besonderen Kenntnisse besitzen, eine Täuschung des Servers zu ermöglichen. Wenn überhaupt ein „Crack“ für ein von Marvin geschütztes System existiert, dann muss es sich um einen vollständigen Re-Rewriter handeln.
Stellt man den nötigen Aufwand und die nötige Sachkenntnis für einen Angriff ins Verhältnis zu dem möglicherweise zu erzielenden Gewinn (der natürlich von der

jeweiligen Anwendung abhängig ist), kann man davon ausgehen, dass die Wahrscheinlichkeit für einen erfolgreichen Angriff sehr gering ist.
Ein generelle Schwierigkeit bei derartigen Sicherheitsüberlegungen besteht allerdings darin, dass sie nur solche Angriffsstrategien berücksichtigen können, die überhaupt im Ansatz vorausgesehen wurden. Es kann immer Angriffe geben, die auf kreative und intelligente Weise Funktionen einer Software in einem Zusammenhang ausnutzen, an den der Autor nie gedacht hatte. Auch können bei der Implementation eines an sich sicheren Konzepts durch Programmfehler Sicherheitslücken entstehen. Und nicht zuletzt kann dies auch als Folge mangelhafter Administration geschehen – unsicherer Umgang mit Passworten oder Fehlkonfiguration von Zugangsrechten sind nur zwei Beispiele, die leider nicht selten anzutreffen sind.

6 Referenzen

6.1 Verwandte Arbeiten

Es gibt im Zusammenhang mit sog. mobilen Agenten Forschungsansätze zur sicheren Ausführung mobilen Codes. Dort wird einerseits untersucht, wie sich mobile Agenten gegenüber nicht vertrauenswürdigen Hosts, auf denen sie ausgeführt werden, absichern können; diese Fragestellung steht in sehr engem Zusammenhang mit dem, was in dieser Arbeit als „sichere Ausführung mobilen Codes" bezeichnet wird. Oft wird auch die Frage gestellt, wie sich solche Hosts, auch als *Computation Server* bezeichnet, gegen bösartige Agenten schützen können (z.B. vor übermässigem Ressourcenverbrauch oder Zugriff auf andere im Server befindliche Agenten). Dieser Aspekt wird in der vorliegenden Arbeit nicht behandelt.

Die verwendeten Techniken variieren. Vielfach wird ebenfalls Obfuskation eingesetzt; zum Teil eingeschränkt auf die Auswertung spezieller mathematischer Funktionen [LOU, SAN]. Andere Autoren verfolgen Ansätze, die ganz ohne Obfuskation auskommen [KAR].

Eine Implementation von On-the-fly-Obfuscation zum Schutz von Applets ist meines Wissens mit Marvin bisher erstmalig realisiert worden.

6.2 Literatur

[LIN] Lindholm, Yellin: The Java Virtual Machine Specification; Addison-Welsey Pub Co 1999; ISBN 0-201-43294-3

[LOU] Loureiro, Molva: Function Hiding Based on Error Correcting Codes; in: Blum, Lee (Hrsg.): Cryptographic Techniques and E-Commerce; Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99), City University of Hong Kong Press 1999

[KAR] Karnik, Tripathi: Security in the Ajanta Mobile Agent System; Department of Computer Science, University of Minnesota 1999 (online: http://citeseer.nj.nec.com/karnik99security.html)

[SAN] Sander, Tschudin: Protecting mobile agents against malicious hosts; in: Vigna (Hrsg.), Mobile Agents and Security, Lecture Notes in Computer Science #1419, Springer 1998

6.3 WWW

[FIG] http://www.fightaidsathome.org

[GAM] http://www.gambas.de

[JAD] http://www.geocities.com/SiliconValley/Bridge/8617/jad.html

[OBF] http://www.drjava.de/obfuscator [auf dem Marvin-Rewriter basierender Standalone-Obfuscator]

[PHA] http://www.phattball.com

[SET] http://setiathome.ssl.berkeley.edu