Software Reverse Engineering

Vorlesung und Übung im Frühjahrssemester 2010 an der Universität Mannheim

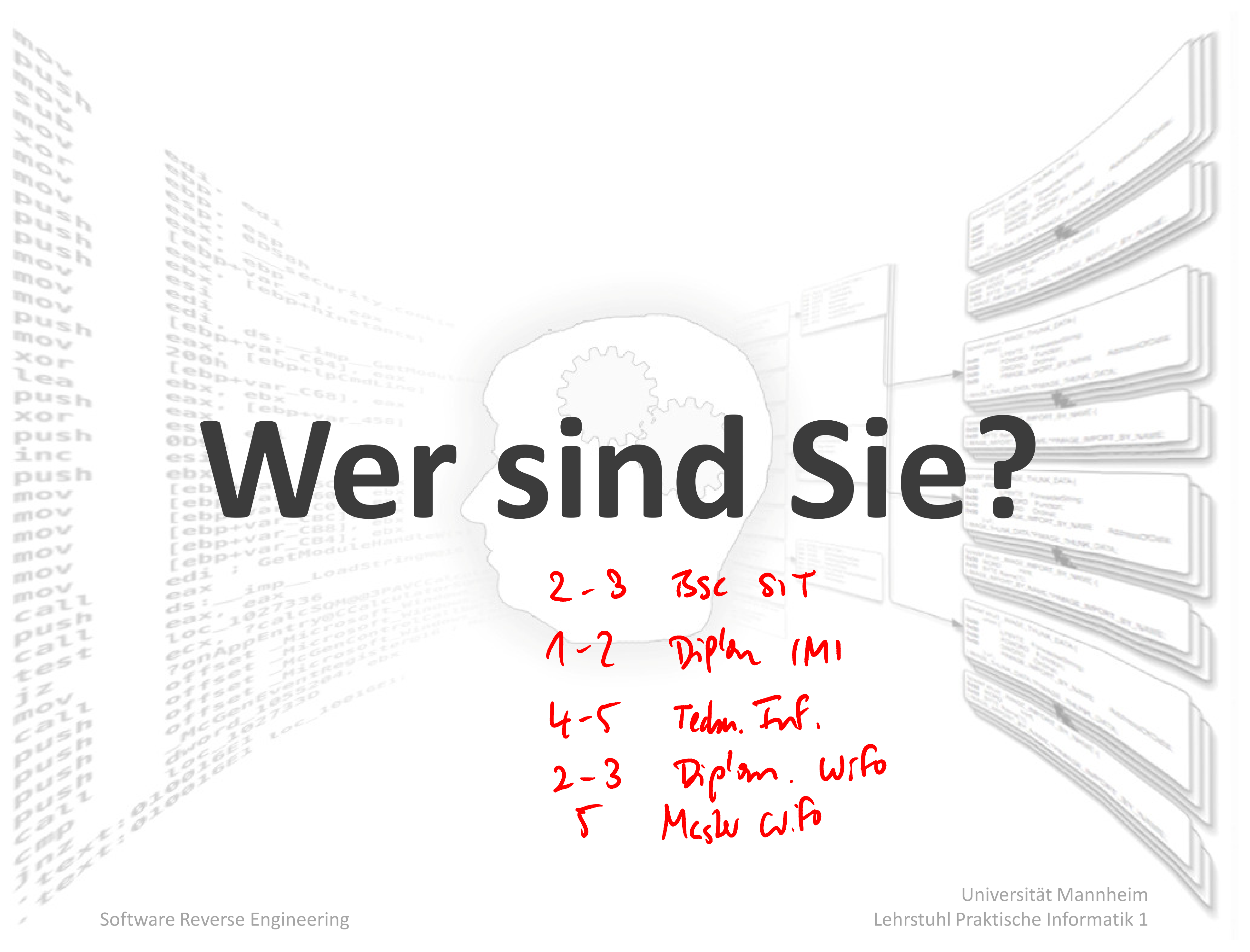
Ralf Hund, Carsten Willems, Felix Freiling

Universität Mannheim
Lehrstuhl Praktische Informatik 1

Kapitel 1: Einführung

Wer sind wir?

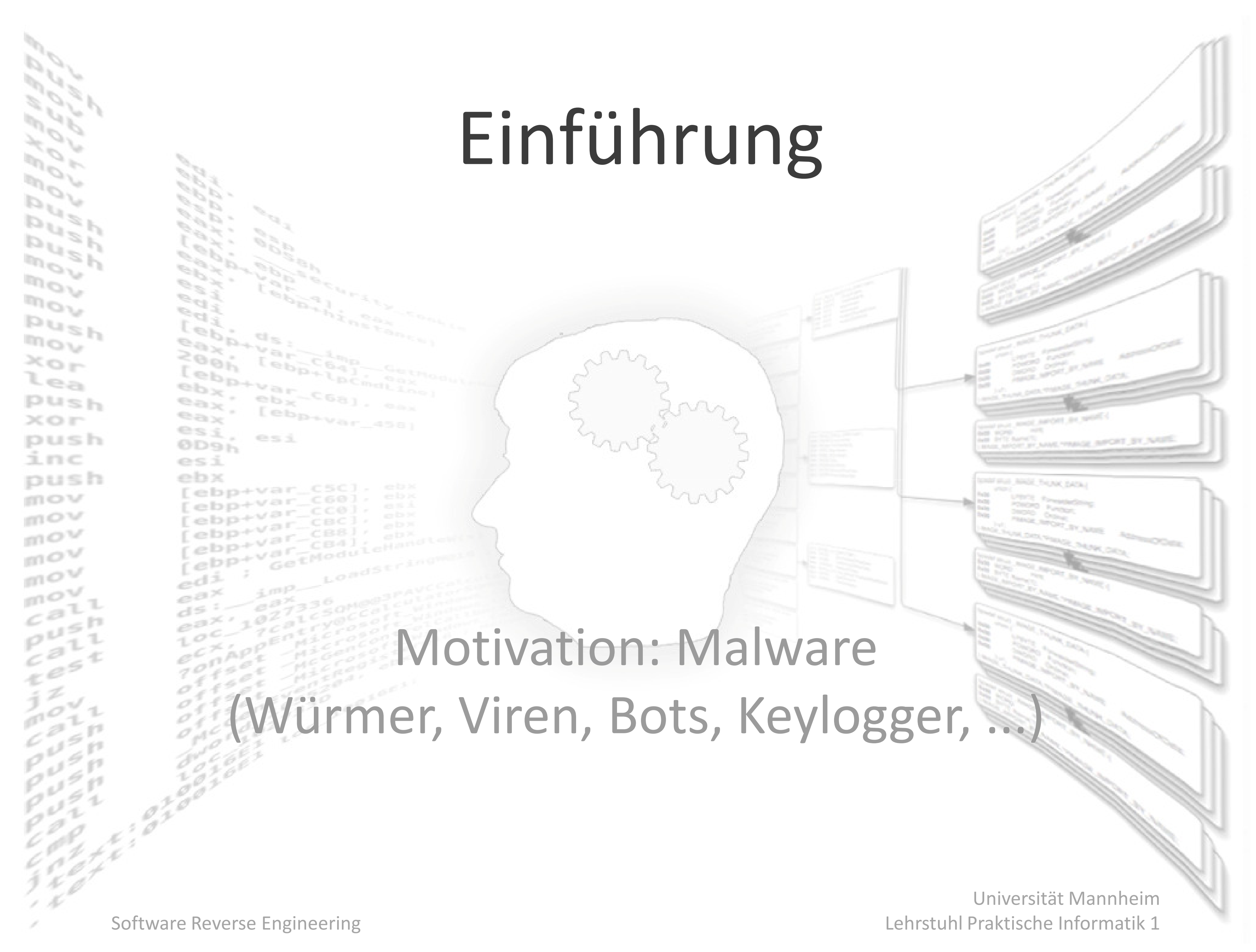
- Felix Freiling
  - Prof. in Mannheim seit 2005
  - Themenfelder: Sicherheit, Betriebssysteme, Forensik
- Ralf Hund
  - Diplom IMI in Mannheim
  - Doktorand bei PI1
  - Extensive Erfahrungen im SRE
- Carsten Willems
  - Diplom Informatik in Aachen
  - Doktorand bei PI1
  - Autor der CWSandbox


Wer sind Sie?
Software Reverse Engineering
Universität Mannheim
Lehrstuhl Praktische Informatik 1

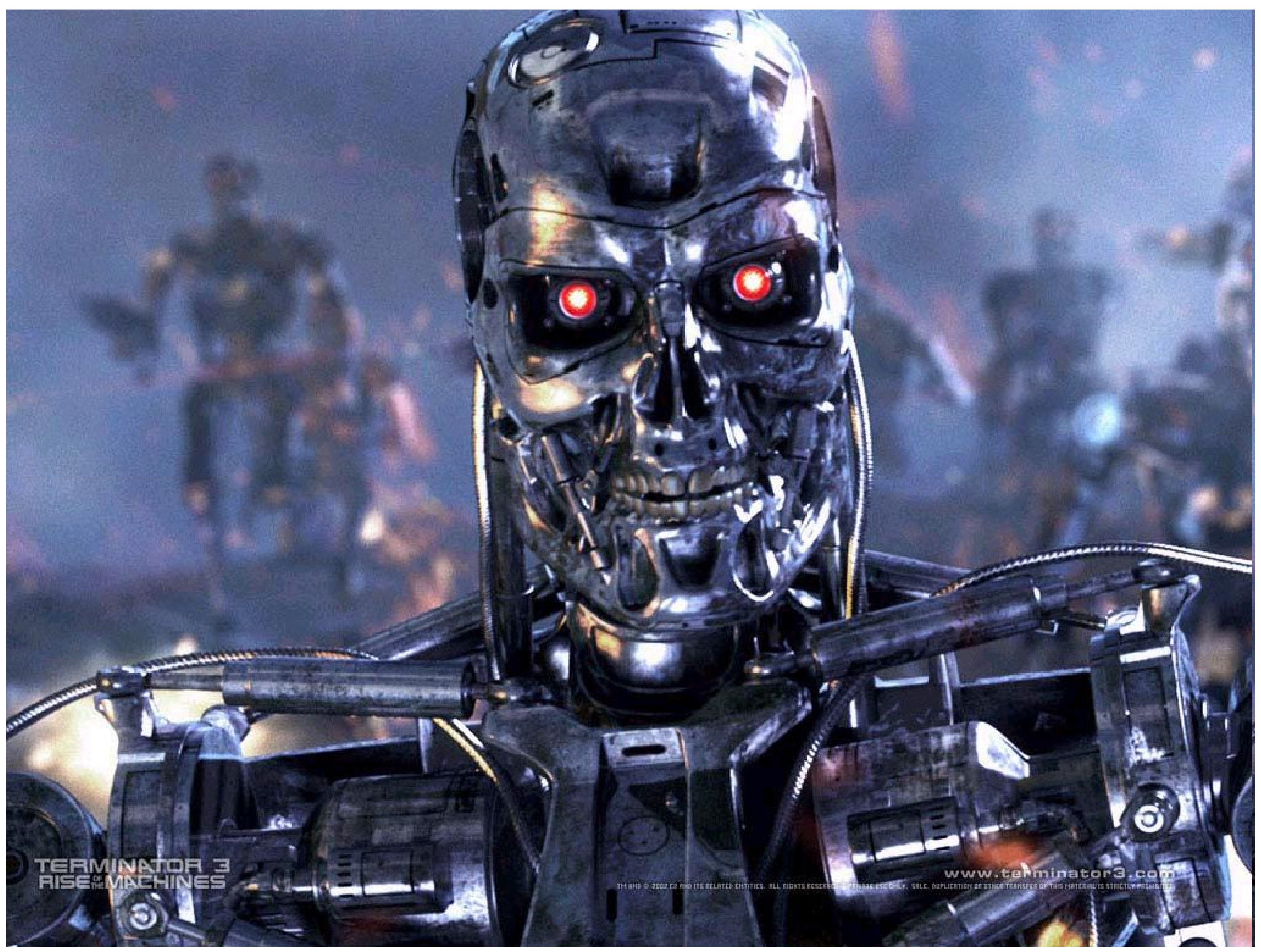
Übersicht

- Motivation: Malware
- Was ist Software Reverse Engineering?
- Grenzen von Software Reverse Engineering
- Formalia: ECTS, Termine, Prüfung
- Ausblick
- Literatur

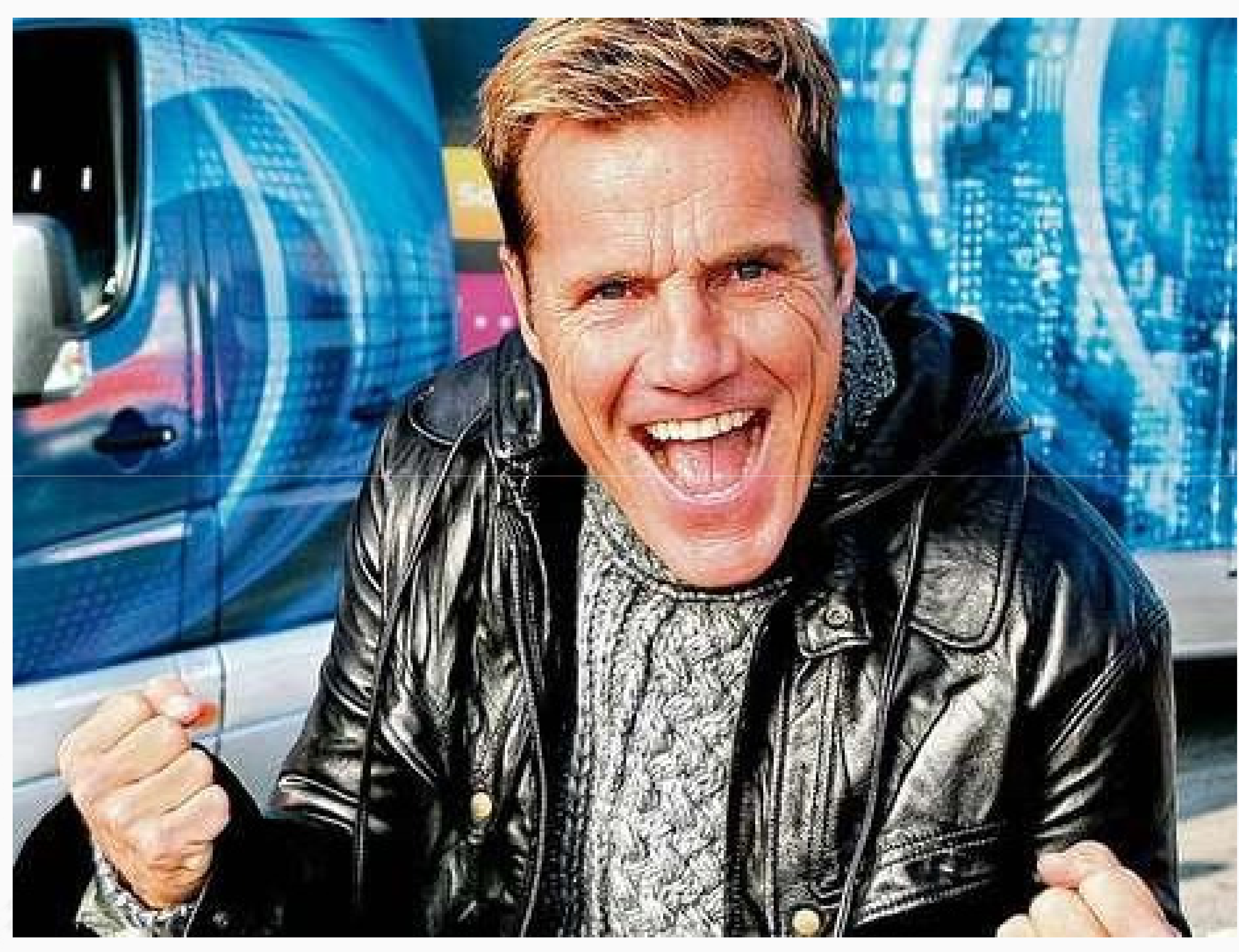
Einführung



Motivation: Malware
(Würmer, Viren, Bots, Keylogger, ...)


TERMINATOR 3
RISE MACHINES
www.terminator3.com

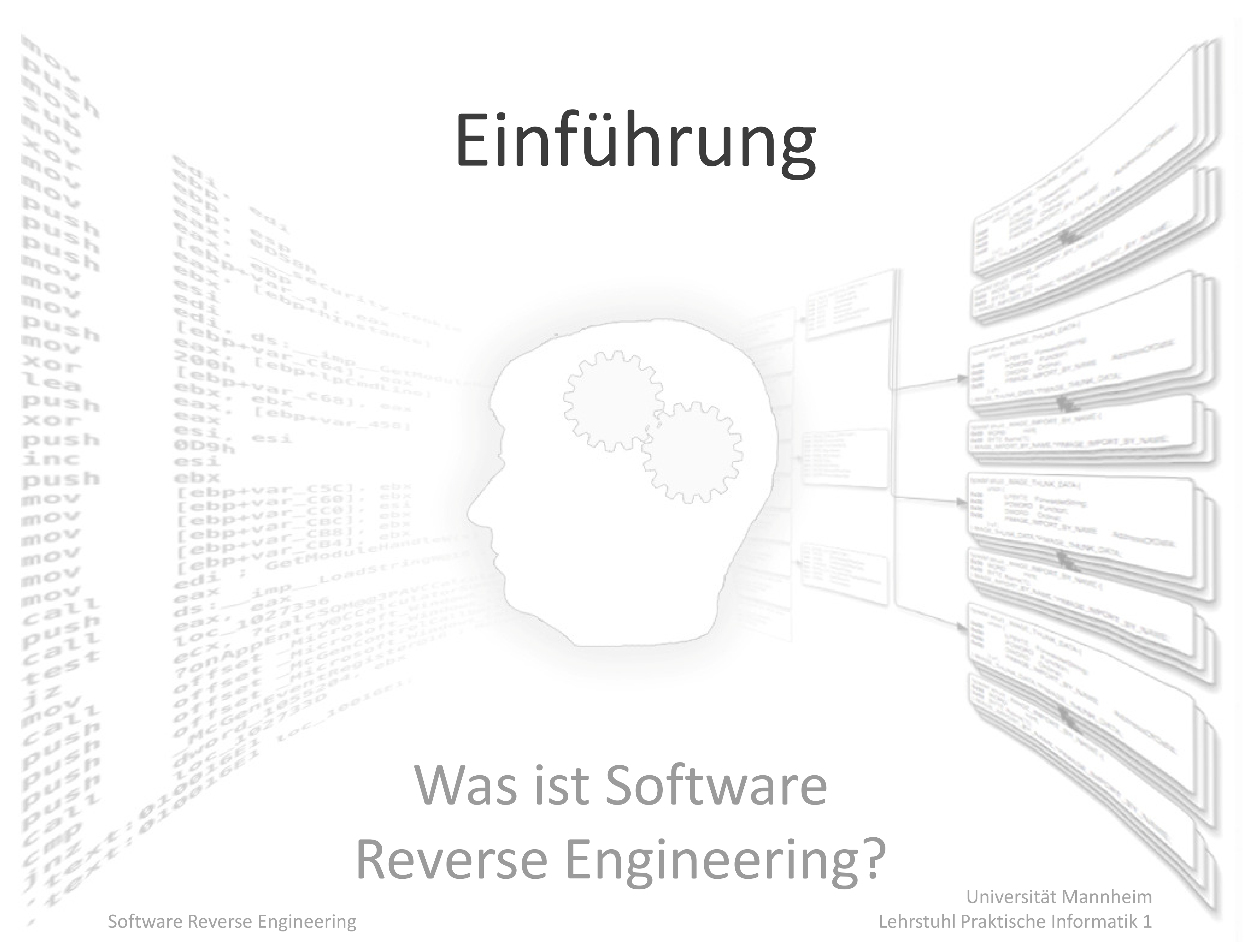
Quelle: IMDB



Quelle: bild.de

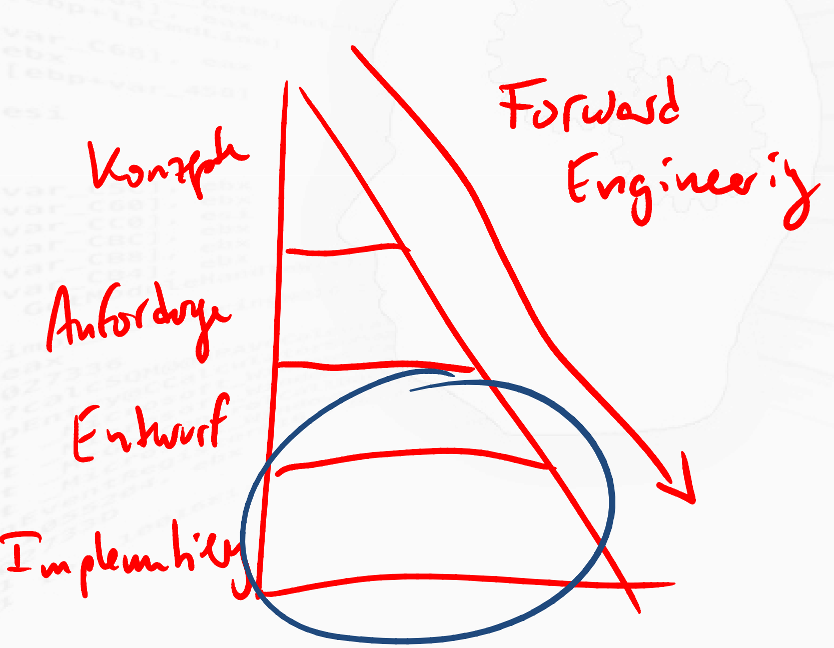
Bezug zur Forensik

- Reverse Engineering ist integraler Bestandteil von vielen forensischen Untersuchungen
  - Digitale Forensik = IT-Beweismittelsicherung und –analyse
  - Viele Straftaten werden mittels Software begangen
- Fragen, die für Ermittlungen relevant sind:
  - Welche Veränderungen hat ein Programm auf einem Rechner potentiell vorgenommen?
  - Mit welchen Rechnern kommuniziert ein Programm potentiell?
  - Welche Daten liest ein Programm, wenn es auf einem Rechner läuft?
  - Wer hat ein bestimmtes Programm geschrieben?
- Bei all diesen Fragen werden Techniken des Software Reverse Engineering benötigt


Einführung
Was ist Software
Reverse Engineering?

Software Engineering

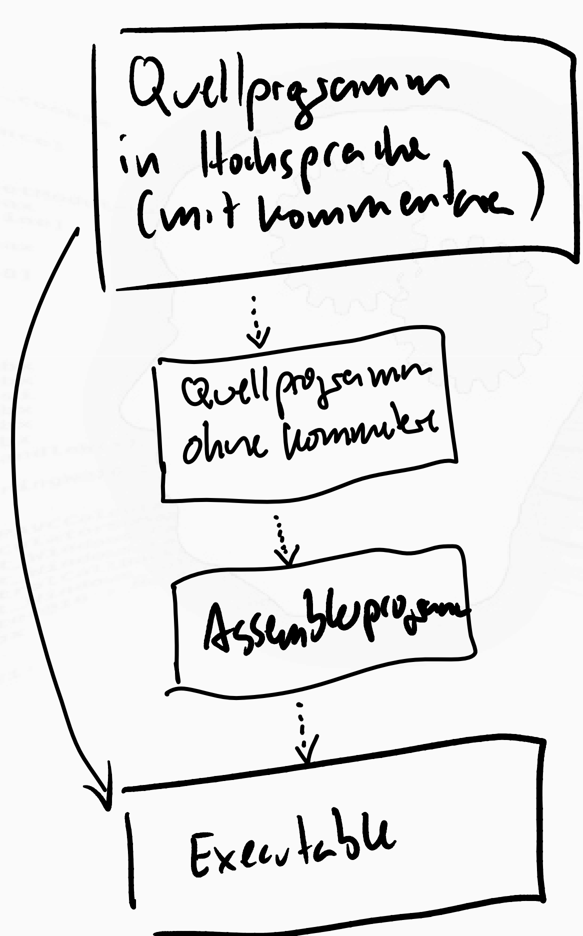
- Strukturierter Prozess zur Entwicklung von Software aus natürlichsprachlich formulierten Anforderungen



Quelle: Byrne 1992

- Mehr: siehe Vorlesung „Software Engineering“

Compilierung

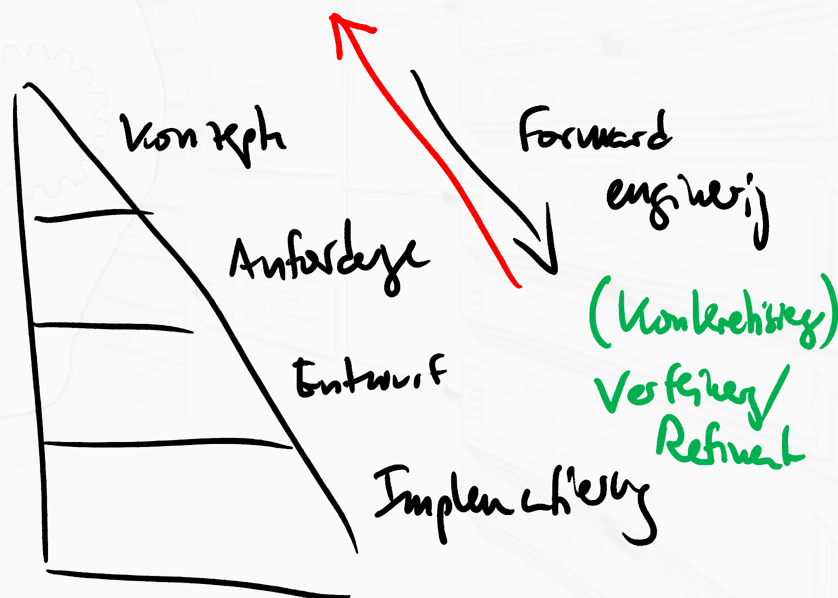


Quelle: Van Emmerik

Software Reverse Engineering

- Umkehrung des Software Engineering Prozesses (Byrne 1992)

Reverse Engineering
(Abstraktion)
Konzepte
Anforderungen
Design
Implementierung



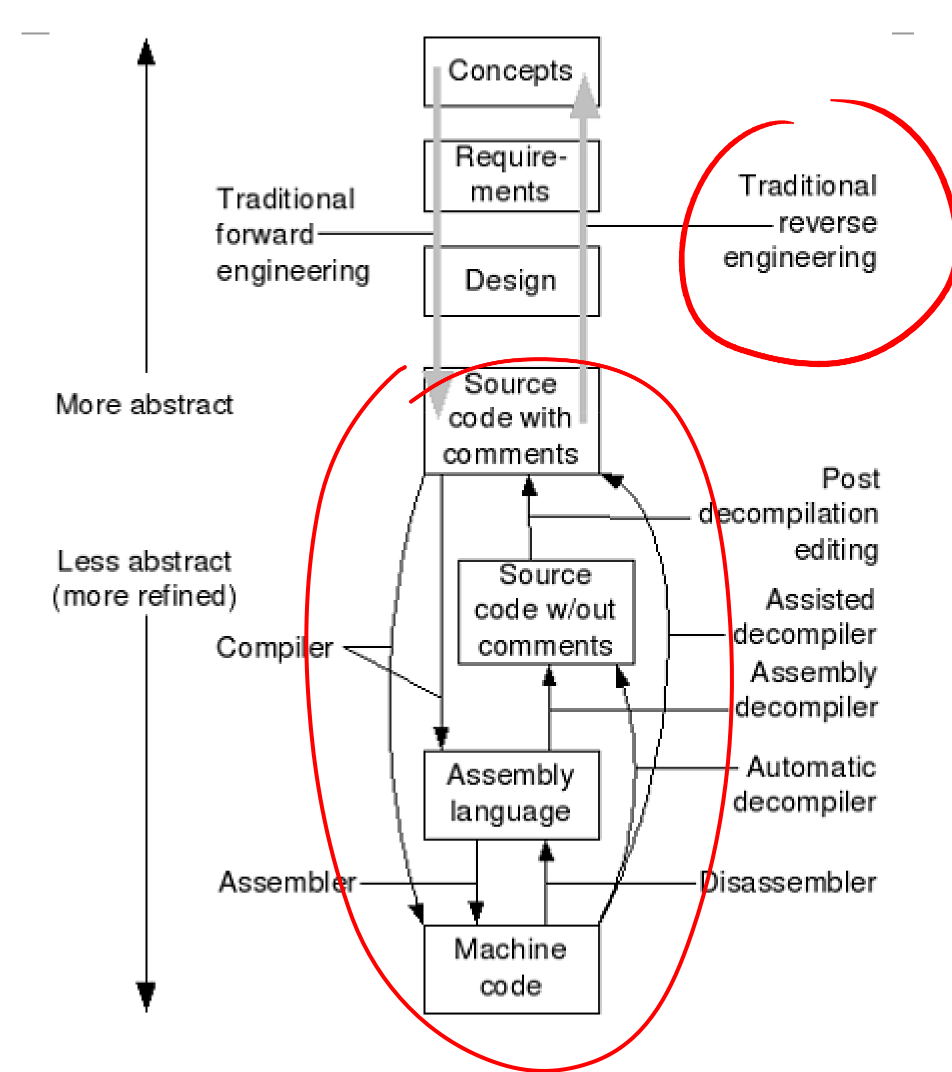
Definition Reverse Engineering

- Definition nach Chikofsky und Cross:
  - The process of analyzing a subject system with two goals in mind:
    1. to identify the system's components and their interrelationships; and,
    2. to create representations of the system in another form or at a higher level of abstraction.

- Analysiertes System bleibt unverändert
  - Ansonsten spricht man auch vom Re-Engineering

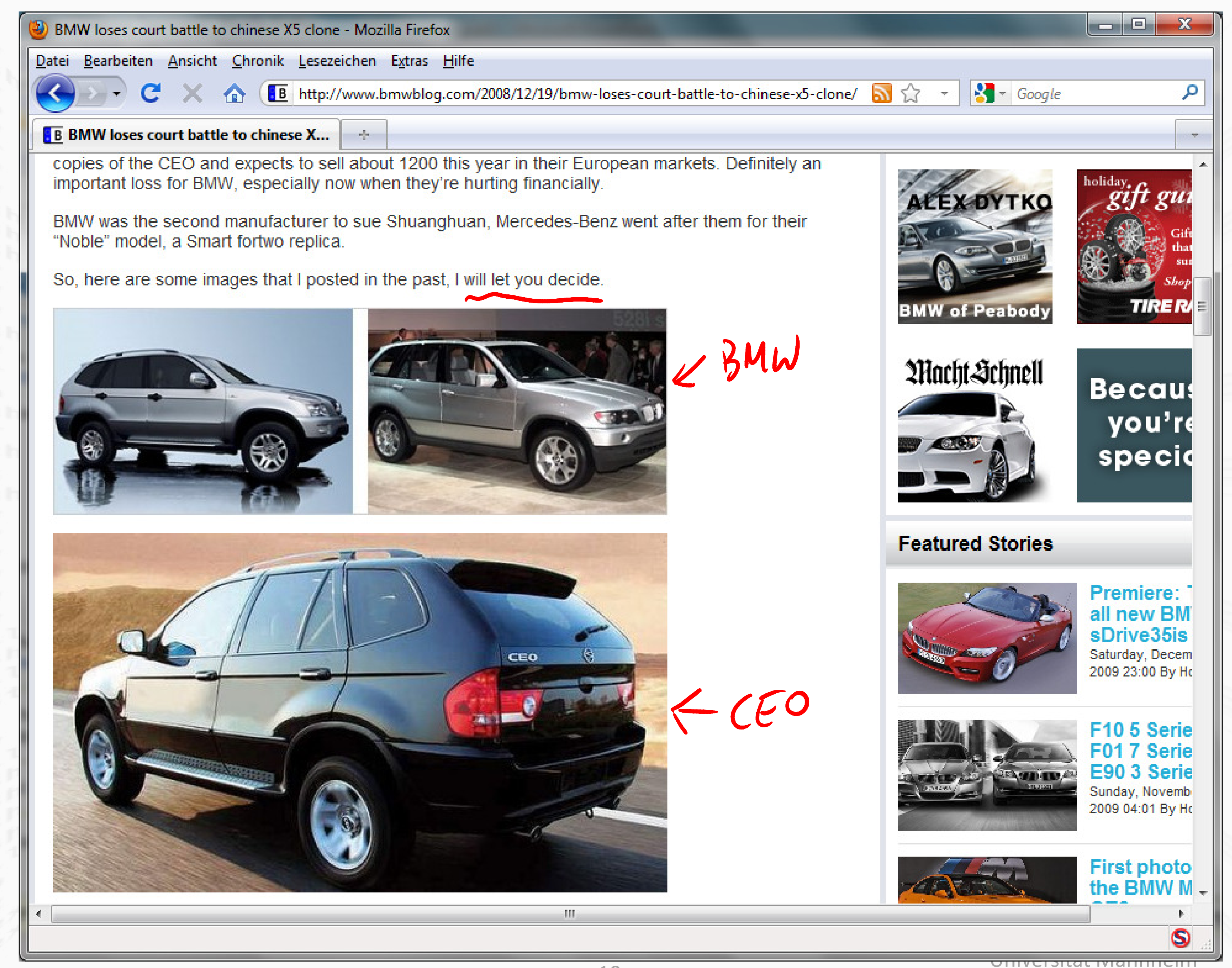
Verfeinerte Darstellung

- Bild setzt bereits Kenntnisse in Compilertechnik voraus
- Traditionell: Fokus auf die Konzepte und Anforderungen
- Heute zusätzlich: Binäranalyse
  - Auch Fokus dieser Vorlesung
- Quelle: Van Emmerik



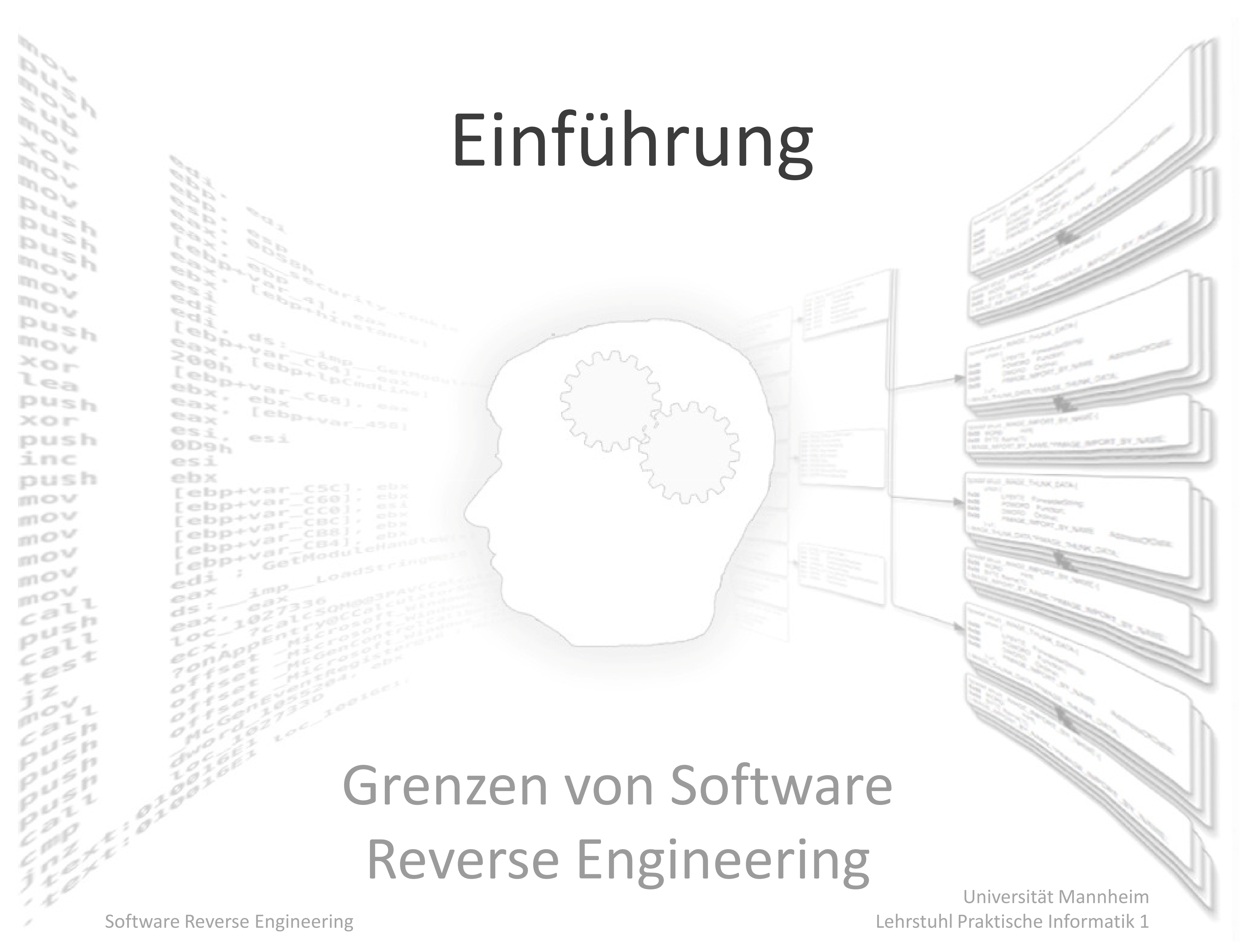
Reverse Engineering

- Software Reverse Engineering vs. Reverse Engineering
  - Reverse Engineering ist das Gegenteil von Forward Engineering
    - the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system [Chikofsky and Cross]
  - Gilt also für alle Arten von Systemen (Maschinen, Autos, etc.)
    - Beispiel: angeblicher BMW X5-Klon von CEO
- Wenn wir "Reverse Engineering" sagen, meinen wir (eigentlich immer) "Software Reverse Engineering"
  - Fokus der Vorlesung: Analyse von Binärcode
  - Eigentlich Decompilierung mit Nachbearbeitung


BMW loses court battle to chinese X5 clone - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://www.bmwblog.com/2008/12/19/bmw-loses-court-battle-to-chinese-x5-clone/
BMW loses court battle to chinese X...
copies of the CEO and expects to sell about 1200 this year in their European markets. Definitely an important loss for BMW, especially now when they're hurting financially.
BMW was the second manufacturer to sue Shuanghuan, Mercedes-Benz went after them for their "Noble" model, a Smart fortwo replica.
So, here are some images that I posted in the past, I will let you decide.
BMW
CEO
ALEX DYTKO
BMW of Peabody
Featured Stories

Anwendungen

- Neben der Analyse von Malware gibt es noch weitere Anwendungsfelder für Reverse Engineering
- Software-Wartung:
  - Bei gewachsenen Systemen gibt es häufig Binärprogramme, zu denen es keine Quellen mehr gibt
    - Oder Programme, die von vornherein in Assembler geschrieben wurden
  - Programme müssen analysiert werden, um sie zu verstehen und gegebenenfalls Fehler zu korrigieren
- Software-Analyse:
  - Software enthält manchmal Geheimnisse, die (finanziell) wertvoll sind
    - Registrierungsschlüssel, Passwörter, Firmengeheimnisse
    - Freischaltung von Vollversionen
  - Legale Implikationen (siehe später in der Vorlesung)



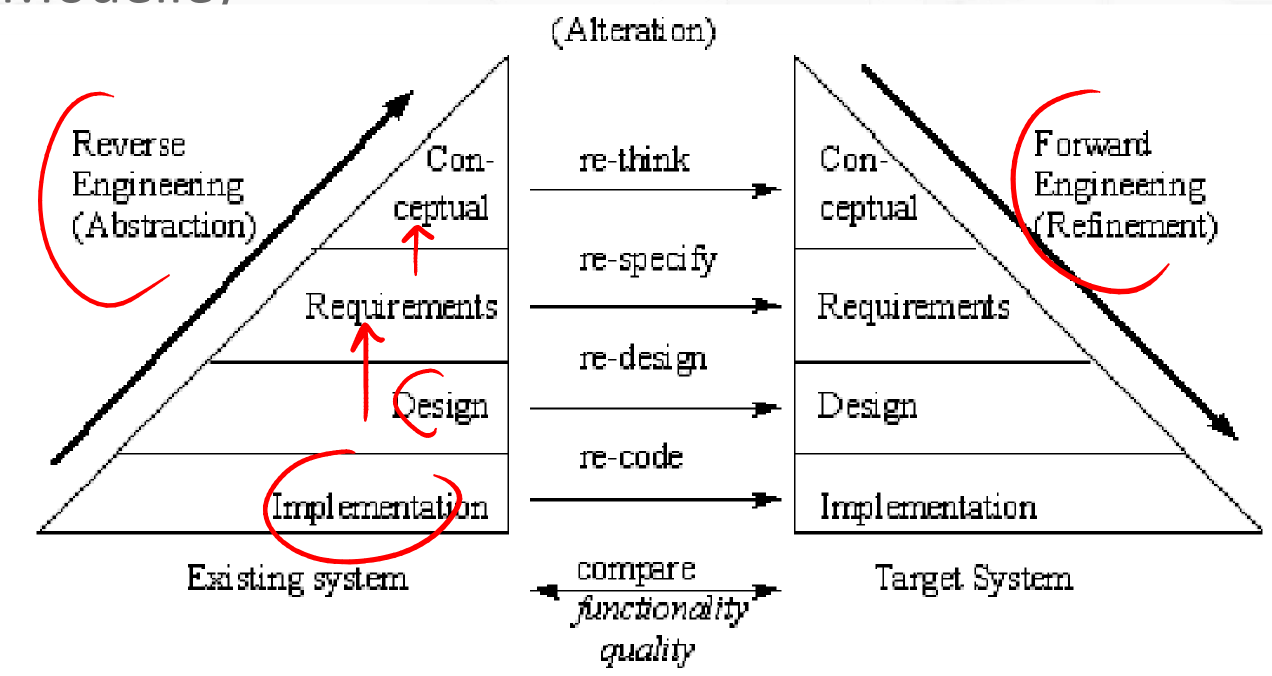
Einführung

Grenzen von Software Reverse Engineering

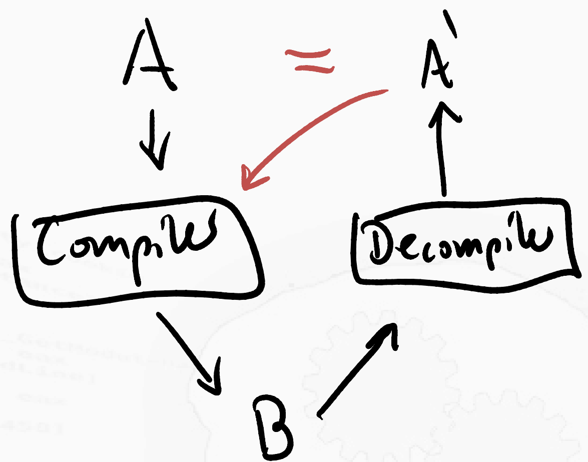
Universität Mannheim
Lehrstuhl Praktische Informatik 1

Traditionelles Reverse Engineering

- Rückübersetzung der Konzepte und Anforderungen immer unscharf
  - Was hat sich jemand gedacht, als er etwas programmierte?
  - Zielt auf informale Anforderungsbeschreibung (Konzepte, Modelle)



Fokus: Decompilierung



- Programm A wird von Compiler in Programm B übersetzt
- Decompiler erzeugt aus B Programm A'
  - Mindestanforderung: erneute Übersetzung von A' soll wieder B ergeben
  - Eigentlicher Wunsch: A' = A

Unterschied Code vs. Daten

- Von Neumann Architektur: keine Unterscheidung zwischen Code und Daten im Speicher
  - Bitmuster im Speicher ist Code, wenn
    - 1) es ein legaler Maschinenbefehl ist und
    - 2) wenn der Befehl irgendwann zur Ausführung kommt.
  - Zufallsmuster enthalten legale Maschinenbefehle
    - Insbesondere bei dichten Binärcodierungen von Maschinensprachen (wie beispielsweise Intel x86)
  - Entscheidende Frage: kommt der Befehl jemals zur Ausführung?

Beispiel

Programm A:

```
01 int i = 50
02 if (i < 1) goto 5
03 i = i - 1
04 goto 2
05 i = 42
```

Programm B:

```
01 int i = 50
02 if (i < 1) goto 5
03 i = i / 1
04 goto 2
05 i = 42
```

- Ist die Zuweisung `i = 42` Code oder kein Code?
  - Wird Zuweisung jemals ausgeführt?

Halteproblem

- Gesucht: Programm H
  - Eingabe: beliebiges Programm P, beliebige Eingabe E
  - Ausgabe:
    - `true` falls P angewendet auf E terminiert (irgendwann anhält)
    - `false` falls P angewendet auf E niemals terminiert

- **Theorem**: Programm H kann es nicht geben („ist nicht berechenbar“)

Intuitiver Beweis

- Annahme: H existiert, Pseudocode:

```
H(P, E) {
  if <P(E) terminiert> return true
  else return false
}
```

- Idee: Wende Programm auf sich selbst an und invertiere Ergebnis
  - Konstruiere H':

```
H'(P) {
  while (H(P, P) == true) ;
}
```

H(H', H')

- Betrachte den Aufruf: H'(H')
  - Führt zu Aufruf H(H', H')
  - Falls Aufruf H'(H') terminiert , dann soll H'(H') nicht terminieren
  - Falls Aufruf H'(H') nicht terminiert, dann soll H'(H') terminieren
  - Widerspruch. H kann also nicht existieren

- Mehr: siehe Vorlesung "Theoretische Informatik"

Unterscheidung Code/Daten

```
while (Bedingung) {
  // Schleifenrumpf
}
AnweisungX
```

- Falls Schleife terminiert, ist `AnweisungX` Code.
  - Ansonsten ist `AnweisungX` kein Code sondern Daten
- Unterscheidung Code/Daten kann auf das Halteproblem reduziert werden
- Schlussfolgerung: Man kann allgemein nicht berechnen, ob eine Speicherzelle Code oder Daten enthält

Selbstmodifizierender Code

- Programme können zur Laufzeit Code-Speicher schreiben
  - Programm wird zur Laufzeit verändert
  - Veränderung kann von Benutzereingabe abhängen
  - Nicht allgemein analysierbar
- Beispiel in x86 Assembler [Cifuentes, S. 3]:
  - **`mov`** schreibt **`0xE920`** an Adresse **`inst`**
  - Verändert den Befehl von **`nop`** (**`0x90`**) in unbedingten Sprung mit Offset **`0x20`** (**`jmp 20`** = **`E9 20`**)

```
        ...                       ; other code
        mov [inst], E920          ; E9 == jmp, 20 == offset
inst db 9090                      ; 90 == nop
```

Idiome

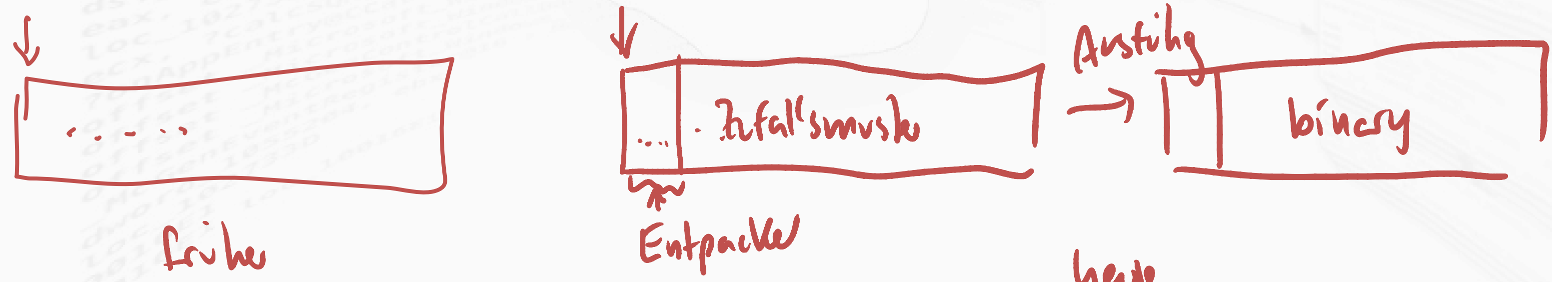
- Idiome sind Programmierkniffe, bei der Instruktionen für einen anderen als ihren offensichtlichen Zweck verwendet werden
- Beispiel:
  - Linksshift um 1 Bit entspricht Multiplikation mit 2
  - Addition zweier Doppelworte durch folgende Befehlskette:
    - Addition der beiden niederwertigen Worte
    - Addition der beiden höherwertigen Worte unter Berücksichtigung des Übertrags aus der ersten Addition
- Idiome muss man kennen, um sie zu verstehen

Generierter Code

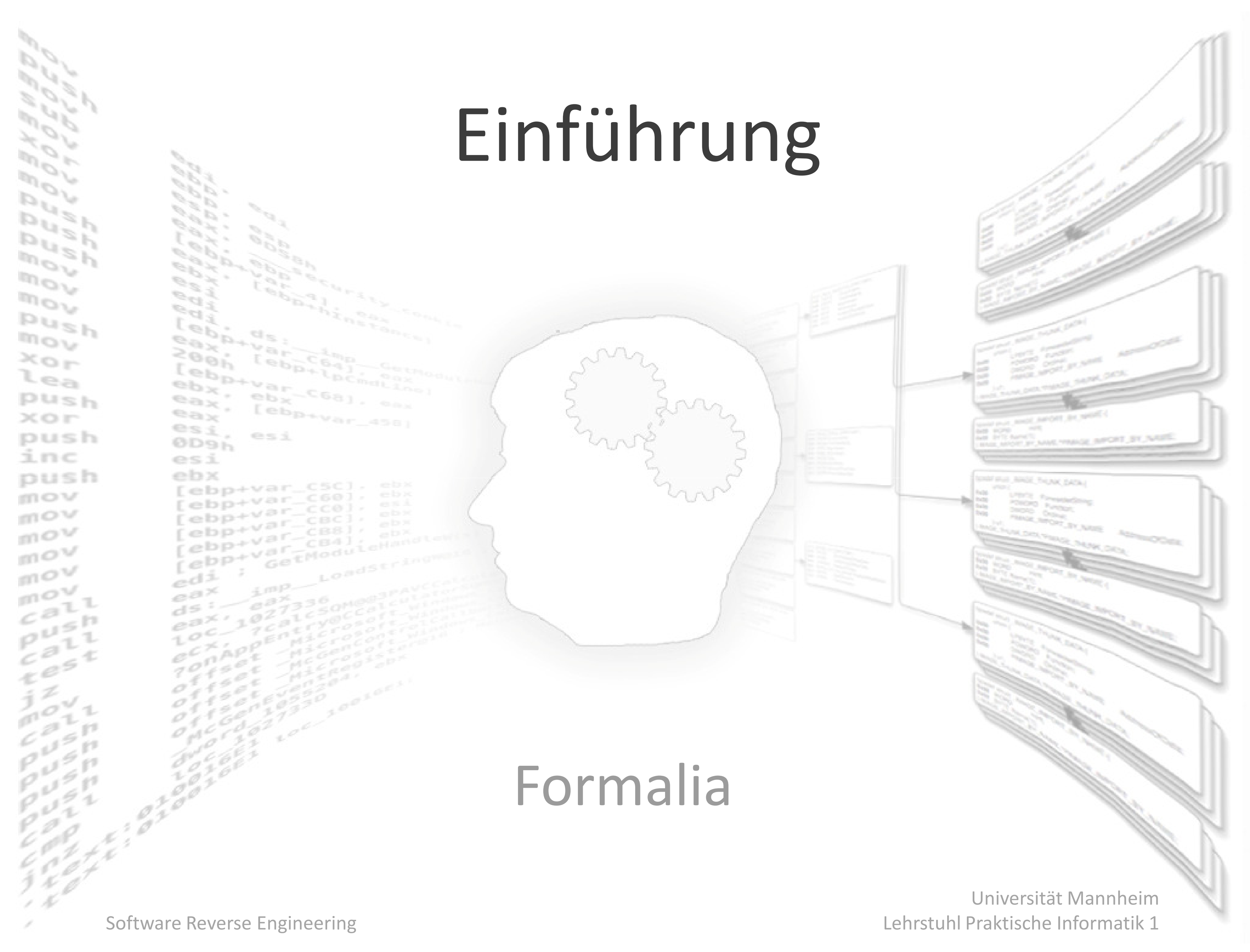
- Beim Übersetzen bauen Compiler oft eigene Codestücke (in Assembler) ein
- Beispiele:
  - Startup-Code, der Bibliotheken einbindet und die Laufzeitumgebung initialisiert
  - Code zur Rettung von Prozessorregistern vor Sprung in eine Subroutine
- Für diesen Code gibt es keine Entsprechung in der Hochsprache

Verschleierung (Obfuscation)

- Code absichtlich kompliziert machen
  - Verschleiert eigentliche Semantik vor einem Menschen
  - Häufig angewandt zur Verhinderung von Reverse Engineering
- Beispiel: gepackte Malware
  - Malware mit vorgeschaltetem Entpacker
    - Eigentliche Malware ist gepackte Payload
    - Packer entpackt Payload zuerst
    - Oft verbunden mit Verschlüsselung
    - Eigentliches Binärprogramm der Malware existiert nur zur Laufzeit



- Mehr Beispiele davon später in der Vorlesung



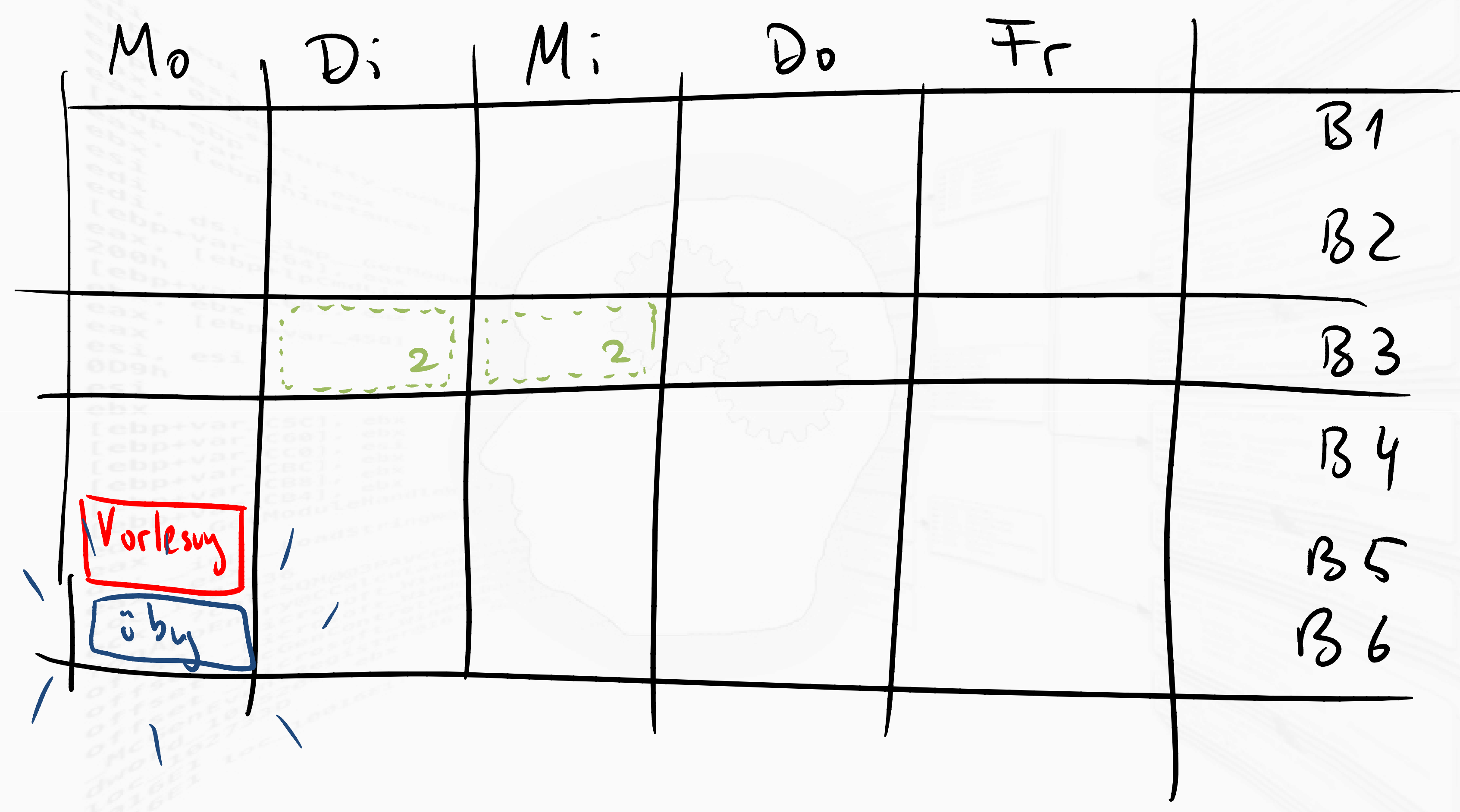
Einführung

Formalia

Formalia

- Vorlesung (V2)
  - Montags, 15:30-17:00 C015
- Übung (Ü2)
  - Montags, 17:15-18:45, ~~verhandelbar~~ beginnt am 22.2.2010
- 6 ECTS
- Verwendbar:
  - Master Wirtschaftsinformatik, Vertiefung
  - Bachelor Wirtschaftsinformatik, Wahlpflicht (auf Antrag)
  - Wahl-/Wahlpflicht Informatik der Diplomstudiengänge
  - Weitere?

Übungstermin



Übungsinhalte

- Beispielhafte Einübung des Stoffes
- Präsentation von Tools
- Besprechung von Hausaufgaben

- Regelmäßiger Termin

- Höhepunkt: selbstständige Analyse eines Malware-Binaries aus der Lehrstuhlsammlung
  - Anfertigen eines kurzen Berichts

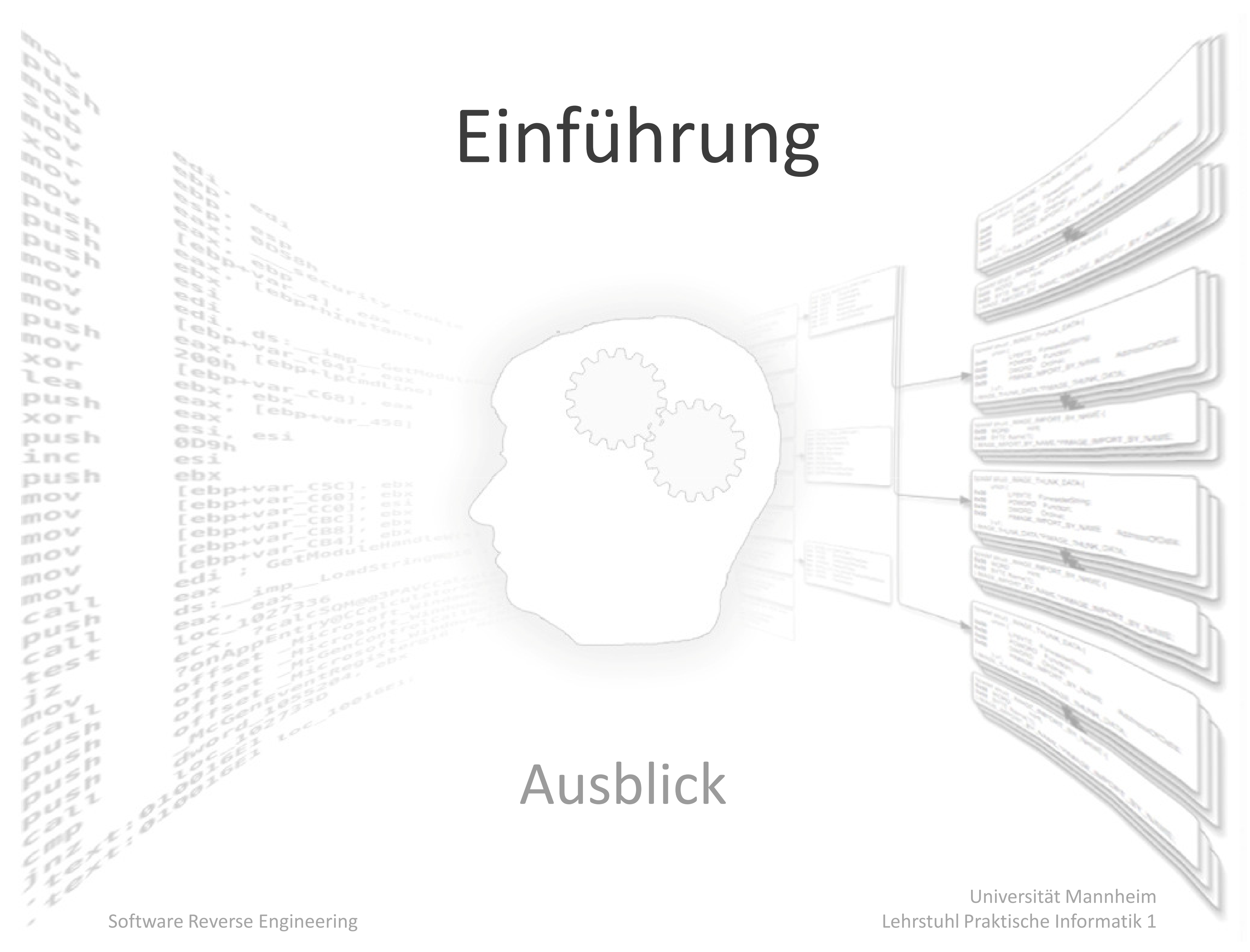
Prüfung

- Prüfung:
  - Entweder Klausur (66 Minuten) oder mündliche Prüfung
  - Entscheidung hängt von Anzahl der Teilnehmer ab und wird vor dem Anmeldungszeitraum gefällt
    - Entscheidung gilt für beide Prüfungstermine
- Übungsteilnahme ist keine formale Voraussetzung für die Prüfung
  - Bearbeitung der Übungen wird dringend empfohlen
  - Bericht über Malware-Analyse kann als Grundlage der Prüfung dienen (bei mündlichen Prüfungen)

Verwandte Lehrveranstaltungen

- Praktische Informatik II
  - Rechnerarchitektur, Maschinensprache
  - in Zukunft auch Compilerbau
- Angewandte IT-Sicherheit
  - Bezug zu Malware, Programmierfehler, Softwaresicherheit
  - Schutzkonzepte
- Betriebssysteme
  - Laufzeitumgebung für Binärprogramme
- Software Engineering
- Weitere?

Einführung



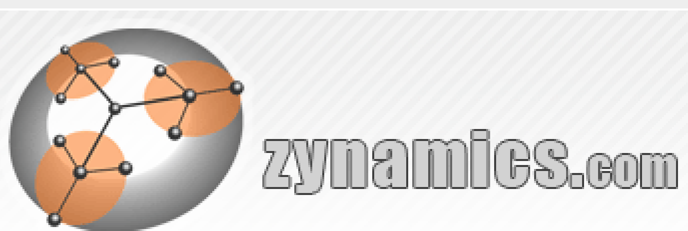
Ausblick

Universität Mannheim
Lehrstuhl Praktische Informatik 1

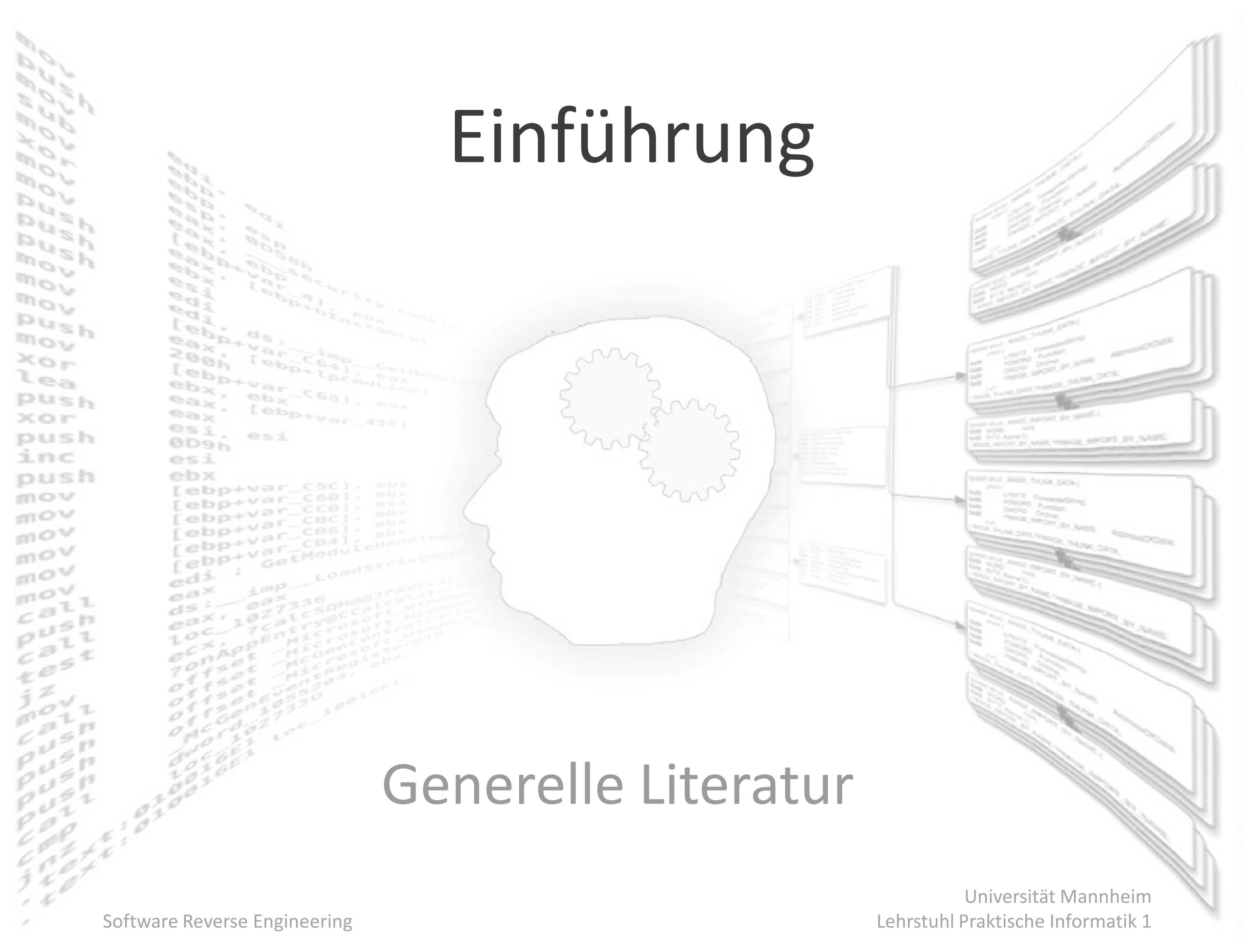
Vorlesungsplan

- Einführung
- Assembler (2 Wochen)
  - Rechnerarchitektur und x86 Assembler
- Hochsprachen
  - Compiler, Linker, Stackframes, Aufrufkonventionen
- Betriebssysteme
  - Adressräume, Paging, System Calls, Prozesse und Threads
- Windows
  - PE-Format, API
- Debugging (Gastvortrag)
  - INT3, Single Stepping, Detektierbarkeit, Debugging API
- Malware (2-3 Wochen)
  - Packer, Obfuscation, Sandboxing
- Plattformübergreifende RE-Methoden (Gastvortrag)
- Rechtliche Fragen (Gastvortrag)

Gastvorträge

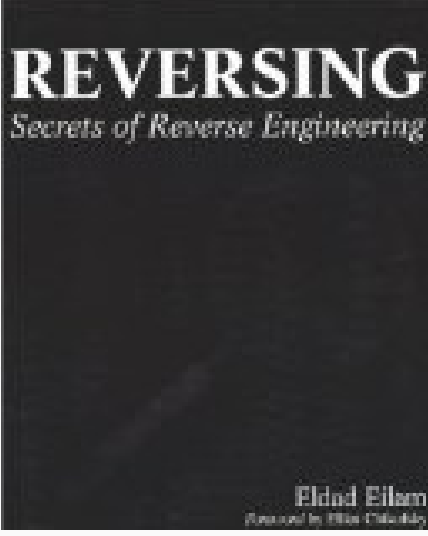


- Sebastian Porst, Zynamics, Bochum
  - Debugging
- Tim Kornau, Zynamics, Bochum
  - Plattformunabhängiges SRE
- Rupert Vogel, Bartsch & Partner, Karlsruhe (angefragt)

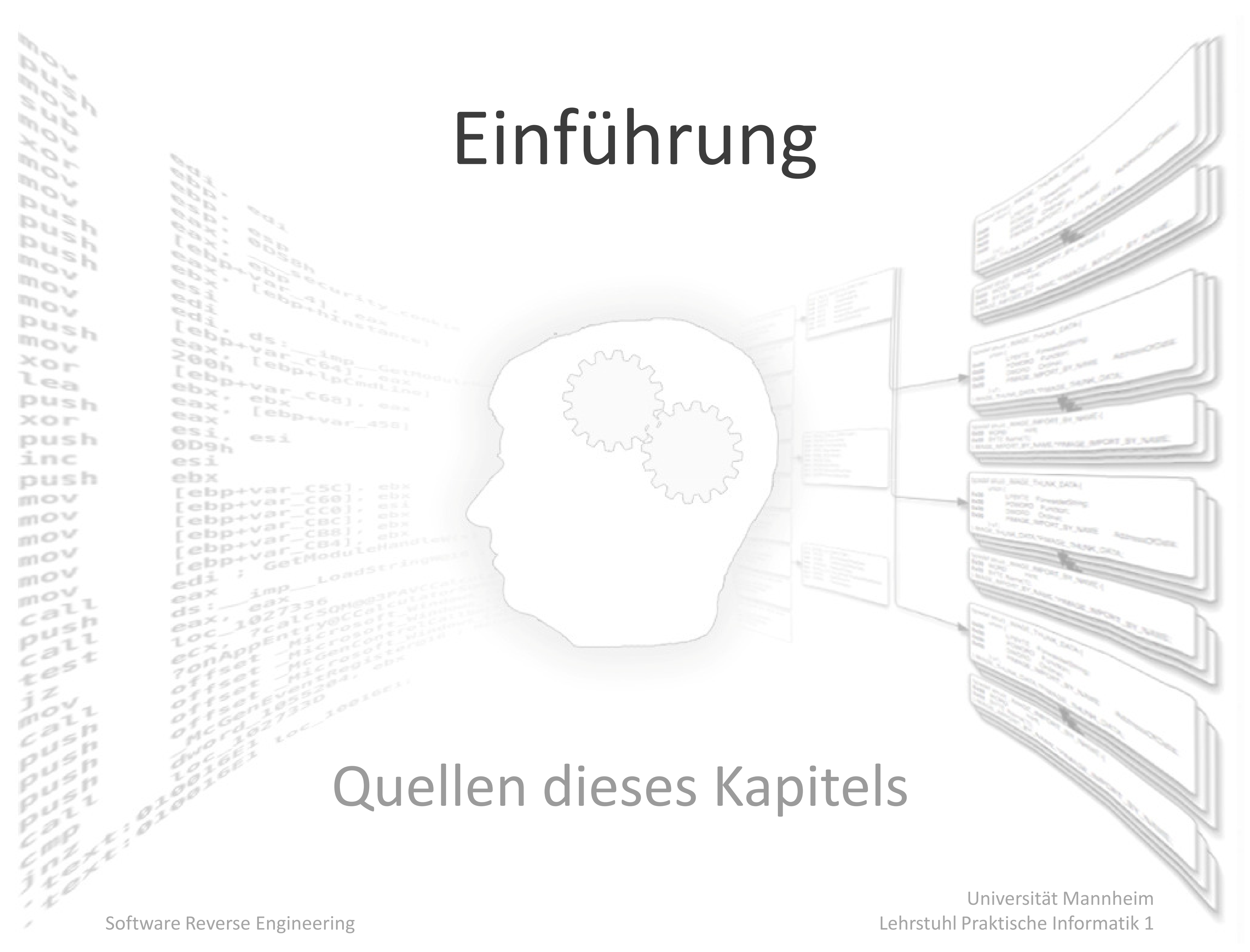

Einführung
Generelle Literatur
Universität Mannheim
Lehrstuhl Praktische Informatik 1
Software Reverse Engineering

Literatur

- Eilam, E.: *Reversing: Secrets of Reverse Engineering*, John Wiley & Sons, 2005
  - Eher praktisch orientiert, deckt Vorlesung nicht vollständig ab



- Spezialliteratur/Quellen werden am Ende des jeweiligen Kapitels angegeben



Einführung

Quellen dieses Kapitels

Quellen dieses Kapitels

- Cristina Cifuentes: Reverse Compilation Techniques. Doktorarbeit, Queensland University of Technology, Australien, Juli 1994.
- Mike van Emmerik: Decompilation and Reverse Engineering. In: Program Transformation Wiki. http://www.program-transformation.org/Transform/DecompilationAndReverseEngineering Stand: 9.2.2010.
- E. J. Byrne: A Conceptual Foundation for Software Re-Engineering. ICSM 1992, pp. 226-235.
- E. Chikofsky, J. Cross: Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1):13-17, 1990