govm: GOvernment Virtual Machine

hc
hc@darmstadt.ccc.de

8. September 2009

ToC

govm: **Go**vernment **V**irtual **M**achine

- Minimalistischer Bytecodeinterpreter + Compiler
- Idee aus GDI3 von Sascha Müller
- Stackmaschine, 16 Opcodes
- Ein- und Ausgabe über *Syscalls*
- Erweiterung: Stackmaschine, 42 Opcodes + BP und SP-Register

govm: **Go**vernment **V**irtual **M**achine

- Minimalistischer Bytecodeinterpreter + Compiler
- Idee aus GDI3 von Sascha Müller
- Stackmaschine, 16 Opcodes
- Ein- und Ausgabe über *Syscalls*
- Erweiterung: Stackmaschine, 42 Opcodes + BP und SP-Register

Stackmaschine

- Beliebt für „idealistische“ VMs
- Alle Operationen auf dem Stack
- Beispiel: 1 + 2 + 3
- Dazu brauchen wir

LI *Load Instant*, Lade Wert vom CS auf den Stack;

ADD *Add*, Hole oberste zwei Werte vom Stack, addiere, lege Ergebnis auf Stack.

Stackmaschine

- Beliebt für „idealistische“ VMs
- Alle Operationen auf dem Stack
- Beispiel: 1 + 2 + 3
- Dazu brauchen wir

LI *Load Instant*, Lade Wert vom CS auf den Stack;

ADD *Add*, Hole oberste zwei Werte vom Stack, addiere, lege Ergebnis auf Stack.

Stackmaschine

- Beliebt für „idealistische“ VMs
- Alle Operationen auf dem Stack
- Beispiel: 1 + 2 + 3
- Dazu brauchen wir

  LI *Load Instant*, Lade Wert vom CS auf den Stack;

  ADD *Add*, Hole oberste zwei Werte vom Stack, addiere, lege Ergebnis auf Stack.

Stackmaschine

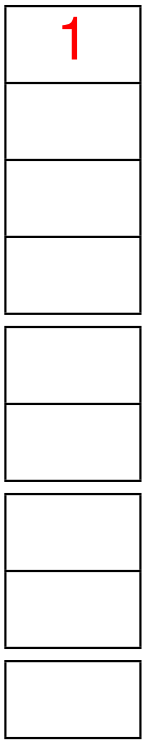
- Beliebt für „idealistische“ VMs
- Alle Operationen auf dem Stack
- Beispiel: 1 + 2 + 3
- Dazu brauchen wir

  LI *Load Instant*, Lade Wert vom CS auf den Stack;

  ADD *Add*, Hole oberste zwei Werte vom Stack, addiere, lege Ergebnis auf Stack.

```
LI  1
LI  2
LI  3
ADD
ADD
```

Stackmaschine



LI 1

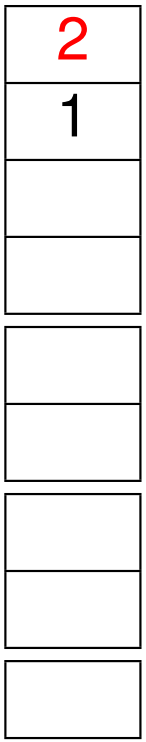
LI 2

LI 3

ADD

ADD

Stackmaschine



LI 1

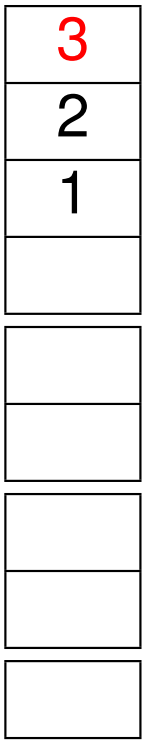
LI 2

LI 3

ADD

ADD

Stackmaschine



LI 1

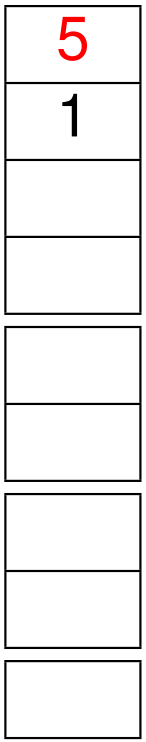
LI 2

LI 3

ADD

ADD

Stackmaschine



LI 1

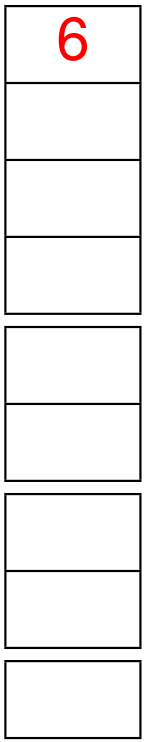
LI 2

LI 3

ADD

ADD

Stackmaschine



LI 1

LI 2

LI 3

ADD

ADD

Syscalls

- Schnittstelle zum System

```
LI 65                # Character to print ('A')
LI 1                 # Syscall number
SYSCALL
```

Syscalls in govm

„Akademisches Subset“:

- getc
- putc
- exit

govm-python-Schnittstelle

```
import govm
import sys

asm = open(sys.argv[1], 'r').read()
def getc():
    chars = list("Hallo Welt")
    while len(chars): yield chars.pop(0)
    yield "\0"
def putc(char):
    sys.stdout.write(char)
    sys.stdout.flush()
res = govm.run(asm, "", "", putc, getc().next)
```

Bytecode-Interpretation (big endian)

```
LI  2
LI  3
ADD
```

CS	1	0	0	0	2	1	0	0	0	3	2
IP	0	1	2	3	4	5	6	8	9	A	B

- IP: Instruction Pointer
- Zeigt auf nächsten auszuführenden Opcode
- IP wird nach jedem Opcode inkrementiert
- Explizites Setzen des IP durch *Sprungbefehle* (JMP, JZ)

Bytecode-Interpretation (little endian)

```
LI  2
LI  3
ADD
```

CS	1	2	0	0	0	1	3	0	0	0	2
IP	0	1	2	3	4	5	6	8	9	A	B

- IP: Instruction Pointer
- Zeigt auf nächsten auszuführenden Opcode
- IP wird nach jedem Opcode inkrementiert
- Explizites Setzen des IP durch *Sprungbefehle* (JMP, JZ)

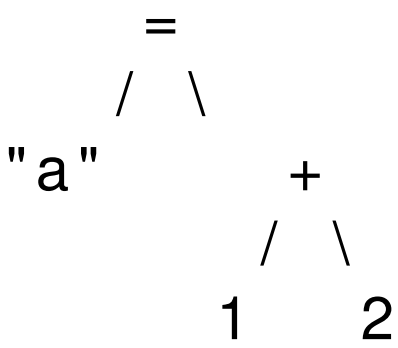
Syntax

```
def main():
    local local_variable:datatype

    function(param1, param2)
    syscall(param1, param2)

    ...
    halt()

def function(param1:datatype, param2:datatype)
    ...
    [return <uint>]
```

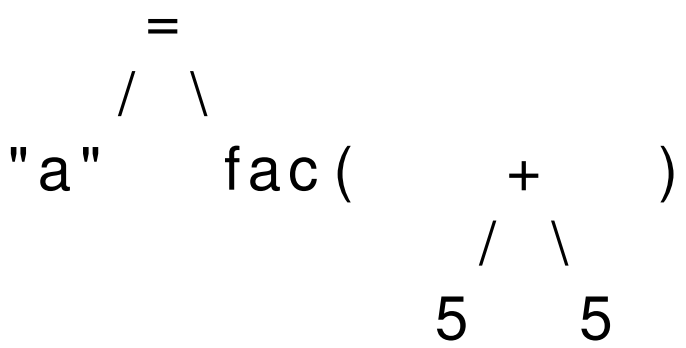
Parsetrees

- (govm:) Hierarchische Darstellung einer Codezeile
- Baumstruktur
- Jede Node:
  - Operator
  - Wert (Integer, String, Objekt)
  - Referenz
  - Funktionsaufruf

a = 1 + 2



a = fac(5 + 5)



Ausführung

- Beginne bei oberster Node
- Enthält Node einen Wert? Zurückgeben.
- Ansonsten: Wert linker und rechter Childnode anfordern
- Dann: Operation ausführen
- Childnode: äquivalentes Vorgehen

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator → Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator → Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator → Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator → Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator → Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator $\rightarrow$ Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

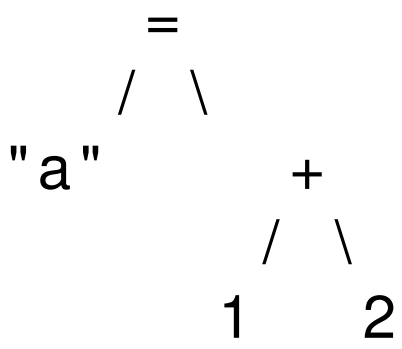
Beispiel: a = 1 + 2

- Oberste Node: '='
- '=' ist Operator: Werte linke, rechte Kindnodes aus
- Linke Kindnode: ist Referenz, gebe selbige zurück an '='
- Rechte Kindnode: ist Operator $\rightarrow$ Werte linke, rechte Kindnode aus
- Beides Werte (1, 2). Gebe diese Zurück
- '+' addiert 1+2, gibt 3 zurück an '='
- '=' weist den Wert 3 der Referenz 'a' zu

Vom Parsetree zum Bytecode

- Bytecode nichthierarchische Repräsentation des Parsetrees
- Vorteil: Effizientere Ausführung
- Umwandlung:
  - Wertnodes: Wert auf den Stack mittels LI
  - Operator-Nodes: Linke, Rechte Childnode auf den Stack, dann: Operator auf den Stack
  - Funktionsaufruf-Nodes:
    - Syscall? Syscall-Aufruf auf den Stack
    - Userdefined Function? LI IP, JMP auf den Stack

Beispiel: a = 1 + 2




```
LI 'a'   # Adresse von a
LI 1     # Wert 1
LI 2     # Wert 2
ADD      # Addiere 1 zu 2
SW       # Resultat in Adresse 'a' speichern
```

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- → Bytecode varriert pro Compilerdurchlauf
- → Keine Identifizierung anhand Länge oder Hash
- → Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- → Bytecode varriert pro Compilerdurchlauf
- → Keine Identifizierung anhand Länge oder Hash
- → Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- → Bytecode varriert pro Compilerdurchlauf
- → Keine Identifizierung anhand Länge oder Hash
- → Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- → Bytecode varriert pro Compilerdurchlauf
- → Keine Identifizierung anhand Länge oder Hash
- → Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- → Bytecode varriert pro Compilerdurchlauf
- → Keine Identifizierung anhand Länge oder Hash
- → Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- $\longrightarrow$ Bytecode varriert pro Compilerdurchlauf
- $\longrightarrow$ Keine Identifizierung anhand Länge oder Hash
- $\longrightarrow$ Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

NOPs

- „Dummy“-Opcode ohne Effekt
- Standard-Anwendungsfall: Warten auf Resultate von mehrere Clockcycles dauernden Operationen (RISC)
- GoVM-Anwendungsfall: Cheatprevention
- govmc baut zufällig NOP-opcodes ein
- $\longrightarrow$ Bytecode varriert pro Compilerdurchlauf
- $\longrightarrow$ Keine Identifizierung anhand Länge oder Hash
- $\longrightarrow$ Identifizierung ausschließlich durch Ausführung (oder subtilere Heuristiken ;-)

Sprünge

- Explizite Änderung des IP
- Unbedingte Sprünge: nicht an Bedingung gebunden
- Bedingte Sprünge: an Bedingung gebunden (IF)
- Rücksprünge: RET (Holt IP vom Stack)

Basepointer

- Pointer auf den „Funktionslokalen“ Stack
- Lokale Variablen relativ zu BP
- Retten und Wiederherstellen bei Funktionsein- bzw. austritt

Basepointer-Illustration

```
def main():
    subroutine()

def subroutine():
    local a:uint
    local b:uint

    a = 1 + 2
```

SS	0	1	SP_{old}	IP_{old} BP	b	a	6	1	2	SP

Funktionsaufrufe

- IP, BP retten (PUSH BP)
- IP an Funktionsanfang setzen (CALL)
- SP setzen (Speicher auf Stack reservieren)
- **Eigentlicher Funktionscode**
- BP wiederherstellen
- Rücksprung, wenn Funktion ausgeführt (RET)

Konditionale Ausführung

- JZ: Springe, wenn vorletzter Stackwert 0

```
LI 1
LI fct1
JZ                  # Sprung erfolgt nicht

LI 0
LI fct2
JZ                  # Sprung erfolgt

fct1:
        # foo
fct2:
        # bar
```

Quelltext

```
def main():
    foo()
    halt()

def foo():
    local a:uint
    local b:uint
    if a == 2:
        b = 3
```

Caller

```
# Rufe Funktion foo() auf

LI -1      # Adresse der Funktion foo
CALL       # IP auf Stack, JMP zu foo

# Nach Rueckkehr von foo()
MOVA       # Wert auf Stack in AX
AMOV       # Wert von AX auf Stack (-> DUP)
```

Callee

```
# Entry-Code der Funktion foo()
FMOV     # BP, SP auf Stack
EMOV
 LI 2    # 2 Subtrahieren
SUB
MOVF     # Und im BP speichern
 LI 3    # 3 WORDS auf dem Stack reservieren (Lokale
SALLOC
```

if a == 1:

```
LI 2   # Adresse von a relativ zu BP
FMOV   # BP auf den Stack
ADD    # Absolute Adresse von a berechnen
LWS    # Wert aus DS[&a] auf den Stack
ROT    # Oberste Stackwerte vertauschen
POP    # Obersten Stackwert verwerfen
LI 1   # 1 auf den Stack
EQU    # Vergleiche beide oberste Stackwerte
LI -1  # Zieladresse
JZ     # Springe wenn oberster Stackwert 0
POP    # Falls nicht gesprungen, Wert vom Stack
```

b = 4

```
LI 3      # Adresse von b
LI 4      # Wert 4
ROT       # Oberste 2 Stackwerte vertauschen (Compile
FMOV      # BP auf Stack
ADD       # BP + &b berechnen
ROT       # Oberste 2 Stackwerte vertauschen
SWS       # SS[BP + &b] = 4
POP POP   # Werte vom Stack
FMOV      # BP auf Stack
LI 2      # Adresse des alten BP auf Stack
ADD       # BP dazuaddieren
MOVE      # Ergebnis in SP,
MOVF      # und BP speichern
JMP 55    # Ruecksprung
```

Komplexeres Beispiel: puts

```
def main():
    puts("Hallo")
    puts("Welt")
    halt()

def puts(s:uint)
    local c:uint

    c = peekb(s)
    while c:
        putc(c)
        s = s + 1
        c = peekb(s)
```

Datenbankanbindung

- Komplette Datenbank von python über getc oder DS übergeben
- Schreiben neuer Datensätze über putc
- Format: key\0value\0key\0value\0\0html output
- govm-bytecode sucht richtigen Datensatz raus
- Einschleusen von Code: Zugriff auf komplette Datenbank

Datenbankanbindung

- Komplette Datenbank von python über getc oder DS übergeben
- Schreiben neuer Datensätze über putc
- Format: key\0value\0key\0value\0\0html output
- govm-bytecode sucht richtigen Datensatz raus
- Einschleusen von Code: Zugriff auf komplette Datenbank

Datenbankanbindung

- Komplette Datenbank von python über getc oder DS übergeben
- Schreiben neuer Datensätze über putc
- Format: key\0value\0key\0value\0\0html output
- govm-bytecode sucht richtigen Datensatz raus
- Einschleusen von Code: Zugriff auf komplette Datenbank

Datenbankanbindung

- Komplette Datenbank von python über getc oder DS übergeben
- Schreiben neuer Datensätze über putc
- Format: key\0value\0key\0value\0\0html output
- govm-bytecode sucht richtigen Datensatz raus
- Einschleusen von Code: Zugriff auf komplette Datenbank

Datenbankanbindung

- Komplette Datenbank von python über getc oder DS übergeben
- Schreiben neuer Datensätze über putc
- Format: key\0value\0key\0value\0\0html output
- govm-bytecode sucht richtigen Datensatz raus
- Einschleusen von Code: Zugriff auf komplette Datenbank

DS Buffer Overflow

- Eher langweilig
- Überschreiben von globalen Variablen
- $\longrightarrow$ Erlangen von Privilegien

DS Buffer Overflow

- Eher langweilig
- Überschreiben von globalen Variablen
- → Erlangen von Privilegien

DS Buffer Overflow

- Eher langweilig
- Überschreiben von globalen Variablen
- $\longrightarrow$ Erlangen von Privilegien

Motivation

- What good is a vm that can't be buffer overflow exploited?
- Einschlägige Buffer-Overflowseiten nutzlos
- $\longrightarrow$ Eigene Attacken
- Buffer Overflow Protection nutzlos

Motivation

- What good is a vm that can't be buffer overflow exploited?
- Einschlägige Buffer-Overflowseiten nutzlos
- → Eigene Attacken
- Buffer Overflow Protection nutzlos

Motivation

- What good is a vm that can't be buffer overflow exploited?
- Einschlägige Buffer-Overflowseiten nutzlos
- $\longrightarrow$ Eigene Attacken
- Buffer Overflow Protection nutzlos

Motivation

- What good is a vm that can't be buffer overflow exploited?
- Einschlägige Buffer-Overflowseiten nutzlos
- $\longrightarrow$ Eigene Attacken
- Buffer Overflow Protection nutzlos

Idee

- $FUNCTION schreibt über „ihren“ Bereich des Stacks hinaus
- Überschreibung der Rücksprungadresse
- Einfach: Rücksprung auf vorhandene Funktion
- Nicht ganz einfach: Rücksprung auf eigenen Code
- Konkrete Beispiele im Beispiel-targz

Idee

- $FUNCTION schreibt über „ihren“ Bereich des Stacks hinaus
- Überschreibung der Rücksprungadresse
- Einfach: Rücksprung auf vorhandene Funktion
- Nicht ganz einfach: Rücksprung auf eigenen Code
- Konkrete Beispiele im Beispiel-targz

Idee

- $FUNCTION schreibt über „ihren“ Bereich des Stacks hinaus
- Überschreibung der Rücksprungadresse
- Einfach: Rücksprung auf vorhandene Funktion
- Nicht ganz einfach: Rücksprung auf eigenen Code
- Konkrete Beispiele im Beispiel-targz

Idee

- $FUNCTION schreibt über „ihren“ Bereich des Stacks hinaus
- Überschreibung der Rücksprungadresse
- Einfach: Rücksprung auf vorhandene Funktion
- Nicht ganz einfach: Rücksprung auf eigenen Code
- Konkrete Beispiele im Beispiel-targz

Idee

- $FUNCTION schreibt über „ihren“ Bereich des Stacks hinaus
- Überschreibung der Rücksprungadresse
- Einfach: Rücksprung auf vorhandene Funktion
- Nicht ganz einfach: Rücksprung auf eigenen Code
- Konkrete Beispiele im Beispiel-targz