

# Ghidra Decompiler



Von

Daniel Ebert

Kurs

Penetration Testing und Computerforensik

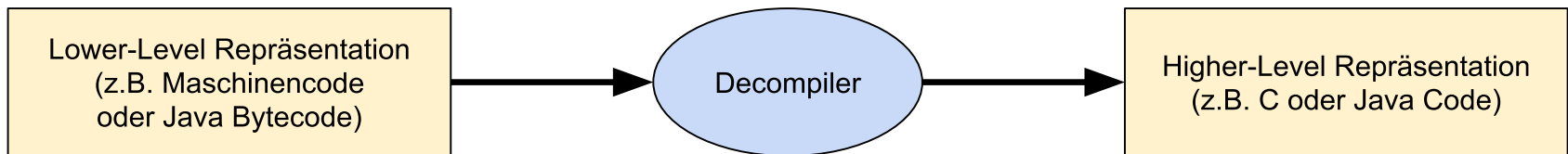
# Übersicht

- Einleitung
- Decompiler Funktionsweise
- Typische Probleme bei der Dekompilation

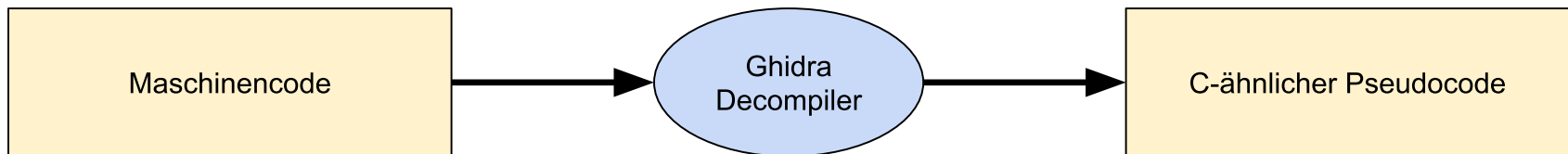
# 1. Allgemeines über Ghidra

- Reverse Engineering Framework
- Komponenten: Decompiler, Plugins, ...
- Entwickelt von NSA, Open Source Community
- 20+ Unterstützte Befehlssatzarchitekturen

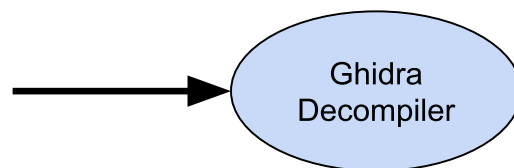
# 1. Decompiler



# 1. Ghidra Decompiler



01101100  
01101111  
01110110  
01100101

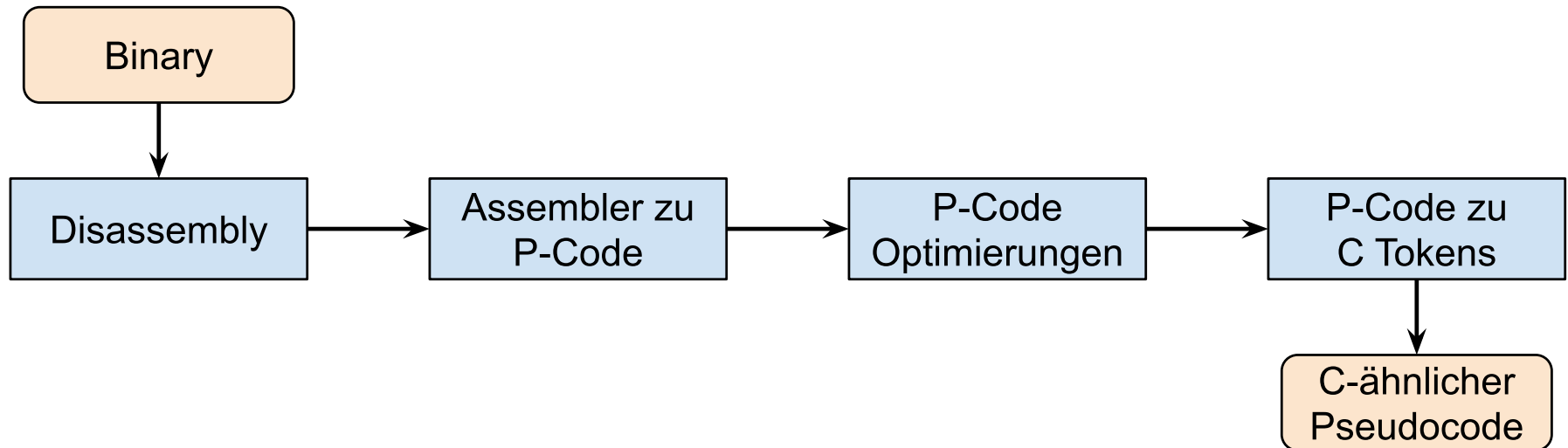


```
Decompile: FUN_00101021 - (noReturn)
1
2 void FUN_00101021(uint param_1)
3
4 {
5     do {
6         printf("x: %d\n", (ulong)param_1);
7     } while( true );
8 }
```

# 1. Decompiler Vorteile

- Programm Funktionsweise einfacher verstehen
- Keine Assemblersprache lernen
- Interaktive Umgebung

## 2. Decompiler Funktionsweise



## 2.1 Disassembly

- Bytes in Daten und Code klassifizieren

55		push	rbp	Ausführbarer Maschinencode
48	89 e5	mov	rbp, rsp	
...		...		
48	01 d0	add	rax, rdx	
3e	ff e0	jmp	rax	
07		(bad)		Daten
00	00	add	%al, (%rax)	
fe		(bad)		
ff		(bad)		
55		push	rbp	Ausführbarer Maschinencode
48	89 e5	mov	rbp, rsp	



# 2.1 Recursive Descent

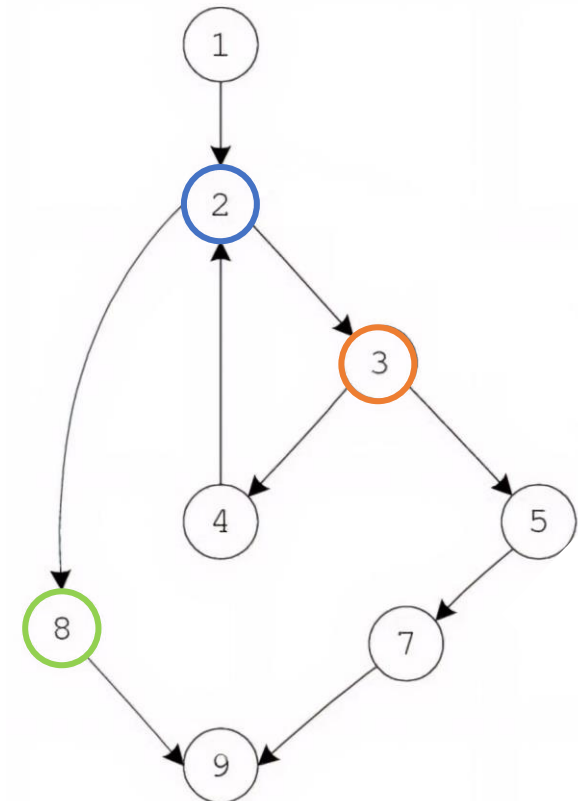
- Kontrollfluss folgen

Basic Block (2)

55	push	rbp
48 89 e5	mov	rbp, rsp
...	...	
48 01 d0	add	rax, rdx
3e ff e0	jmp	rax

Basic Block (8)

55	push	rbp
48 89 e5	mov	rbp, rsp



Basic Block (3)

...

## 2.1 Disassembly

- Statische Ermittlung aller Sprungziele schwierig
  - Heuristik für typische Funktionsprologe
  - Falsche Klassifizierungen
- 
- Übersetzung von Maschinencode zu Assembler

## 2.2 P-Code

- Übersetzung Assembler zu P-Code
- P-Code Operationen
- Input und Output von Operationen sind Varnodes
- Varnodes sind virtuelle Register/Speicher

## 2.2 P-Code Operationen

[COPY](#)  
[LOAD](#)  
[STORE](#)  
[BRANCH](#)  
[CBRANCH](#)  
[BRANCHIND](#)  
[CALL](#)  
[CALLIND](#)  
[USERDEFINED](#)  
[RETURN](#)  
[PIECE](#)  
[SUBPIECE](#)  
[INT\\_EQUAL](#)  
[INT\\_NOTEQUAL](#)  
[INT\\_LESS](#)  
[INT\\_SLESS](#)  
[INT\\_LESSEQUAL](#)  
[INT\\_SLESSEQUAL](#)  
[INT\\_ZEXT](#)  
[INT\\_SEXT](#)

[INT\\_ADD](#)  
[INT\\_SUB](#)  
[INT\\_CARRY](#)  
[INT\\_SCARRY](#)  
[INT\\_SBORROW](#)  
[INT\\_2COMP](#)  
[INT\\_NEGATE](#)  
[INT\\_XOR](#)  
[INT\\_AND](#)  
[INT\\_OR](#)  
[INT\\_LEFT](#)  
[INT\\_RIGHT](#)  
[INT\\_SRIGHT](#)  
[INT\\_MULT](#)  
[INT\\_DIV](#)  
[INT\\_REM](#)  
[INT\\_SDIV](#)  
[INT\\_SREM](#)  
[BOOL\\_NEGATE](#)  
[BOOL\\_XOR](#)  
[BOOL\\_AND](#)

[BOOL\\_OR](#)  
[FLOAT\\_EQUAL](#)  
[FLOAT\\_NOTEQUAL](#)  
[FLOAT\\_LESS](#)  
[FLOAT\\_LESSEQUAL](#)  
[FLOAT\\_ADD](#)  
[FLOAT\\_SUB](#)  
[FLOAT\\_MULT](#)  
[FLOAT\\_DIV](#)  
[FLOAT\\_NEG](#)  
[FLOAT\\_ABS](#)  
[FLOAT\\_SQRT](#)  
[FLOAT\\_CEIL](#)  
[FLOAT\\_FLOOR](#)  
[FLOAT\\_ROUND](#)  
[FLOAT\\_NAN](#)  
[INT2FLOAT](#)  
[FLOAT2FLOAT](#)  
[TRUNC](#)  
[CPOOLREF](#)  
[NEW](#)

[1]

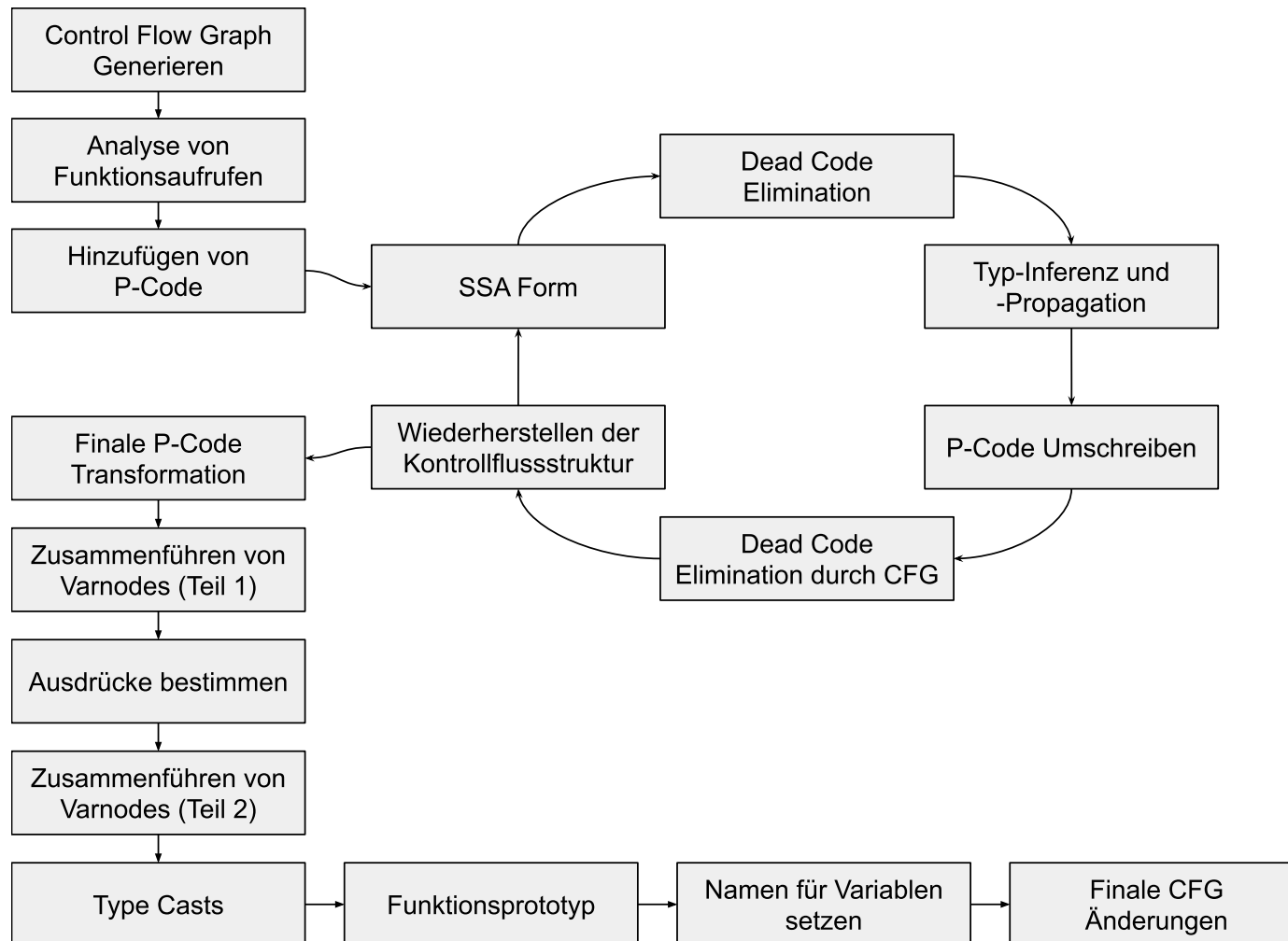
## 2.2 P-Code Vorteile

- Architekturneutral
- Algorithmen für nur 1 Sprache
- Designed für Analyse und Optimierungen
- Keine Side-Effects

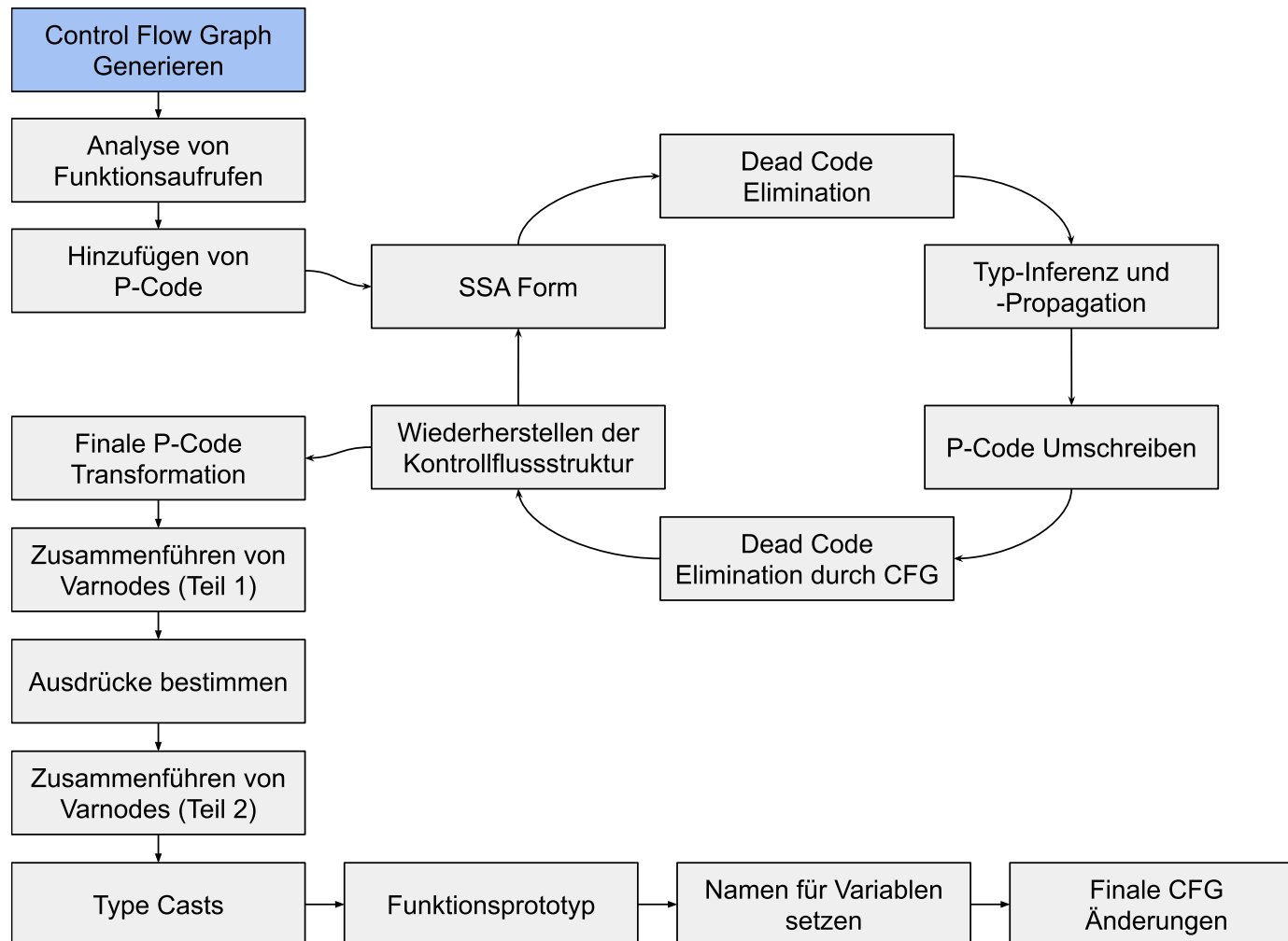
## 2.3 Ziel der P-Code Optimierung

- Optimieren nach Lesbarkeit
- Higher-Level Kontrollflussstrukturen (Schleifen, ...)
- Datentypen

## 2.3 P-Code Optimierung

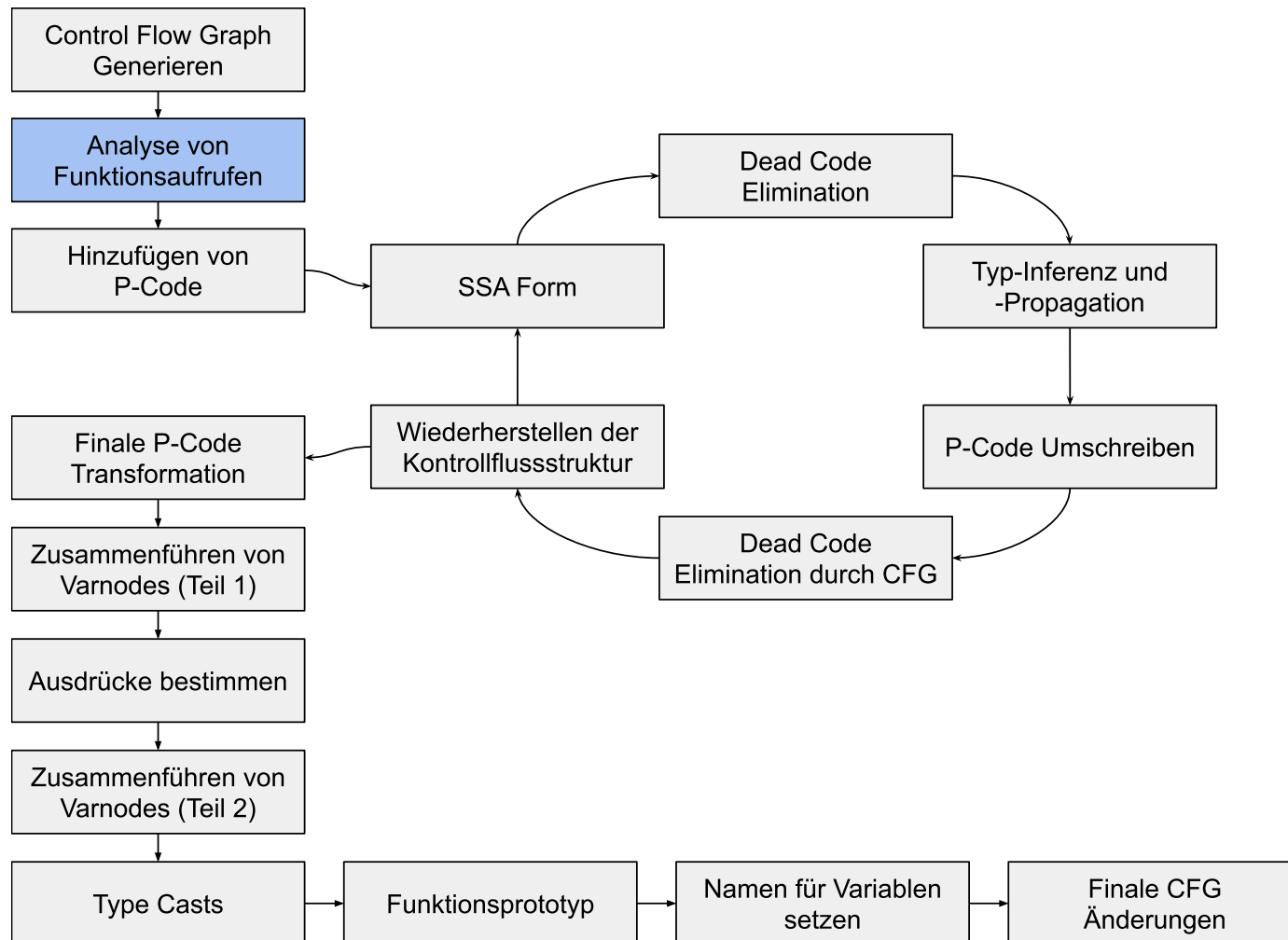


## 2.3 Control Flow Graph



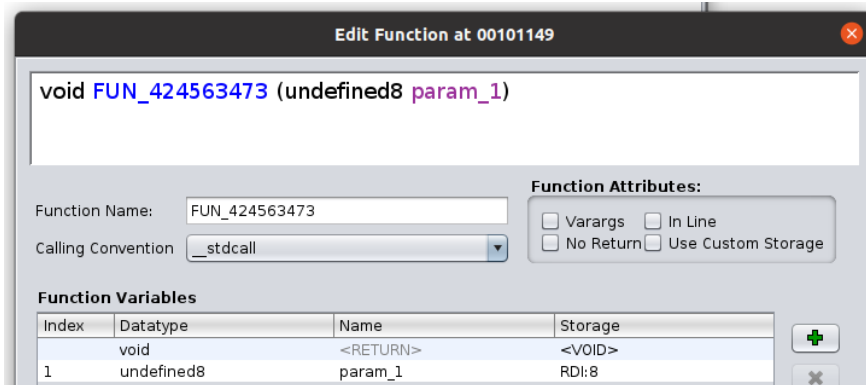


## 2.3 Analyse von Funktionsaufrufen



## 2.3 Funktion ID

- Funktionsname, # Parameter, Parametertypen, Rückgabebetyp, Calling Convention, ...



```
Decompile: FUN_424563473 - (print)
1
2 void FUN_424563473(undefined8 param_1)
3
4 {
5     printf("current_speed: %lu", param_1);
6 }
```

## 2.3 Funktion ID Database

- Funktionssignaturen für bekannte Libraries
- Nur für statisch gelinkte Binaries

### Ghidra Function ID dataset repository

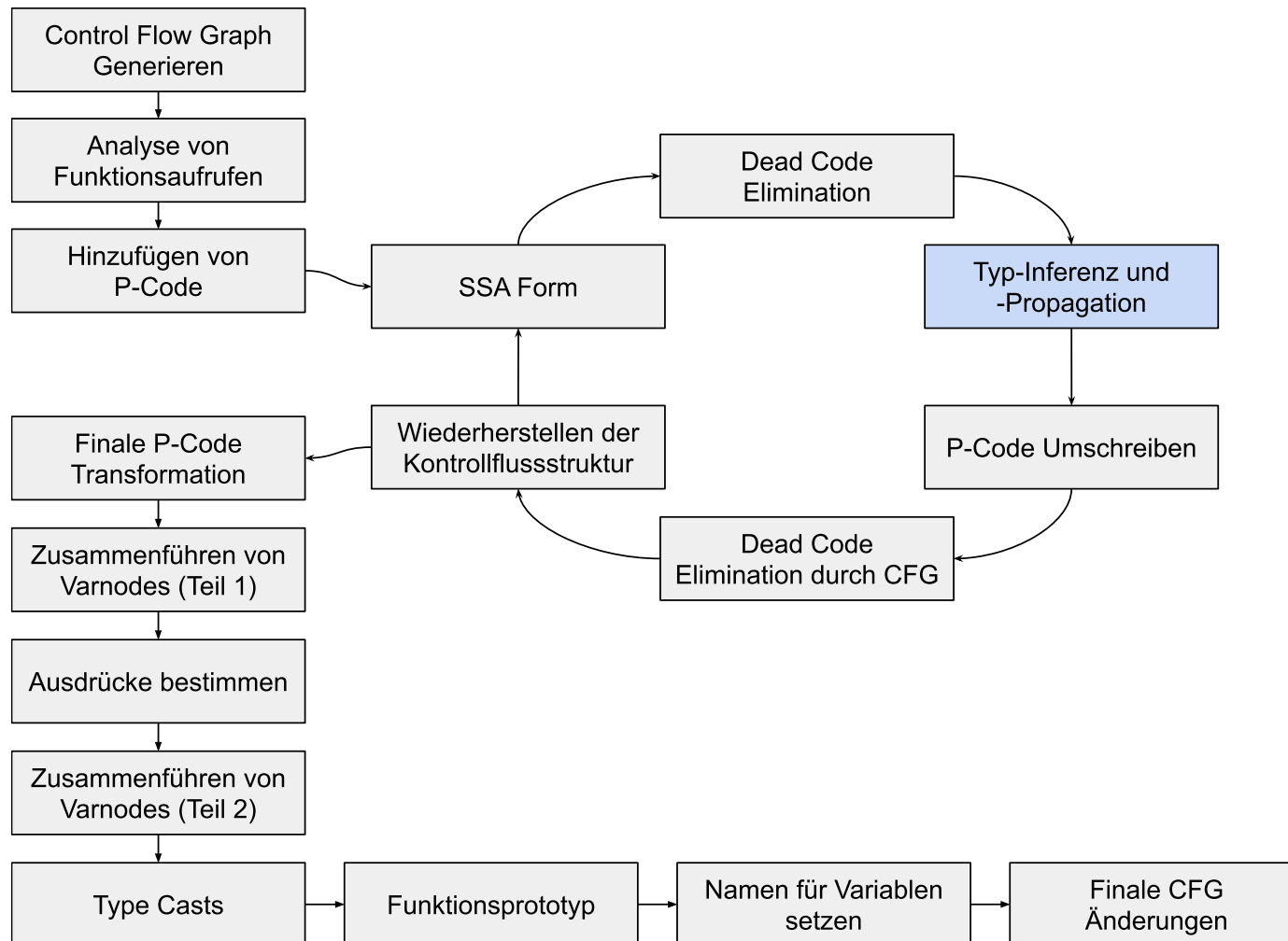


#### Content

- `ubuntu (libc, ...)` : <http://de.archive.ubuntu.com/ubuntu/pool/main/>
- `sigmoid` : openssl libraries from <https://www.npcglib.org/~stathis/downloads/>
- `libsodium` : <https://download.libsodium.org/libsodium/releases/>
- `el{6,7}` : <http://mirror.centos.org/centos/>
- `gcc` : from `el{6,7}` and `ubuntu`
- `teskalabs` : <https://teskalabs.com/blog/openssl-binary-distribution-for-developers-static-library>

[3]

## 2.3 Typ-Inferenz und -Propagation



## 2.3 Typ-Inferenz und -Propagation

- Analysiert Kontext in dem Variable verwendet wird
- Typen propagiert durch COPY, LOAD, STORE

### FLOAT\_ADD

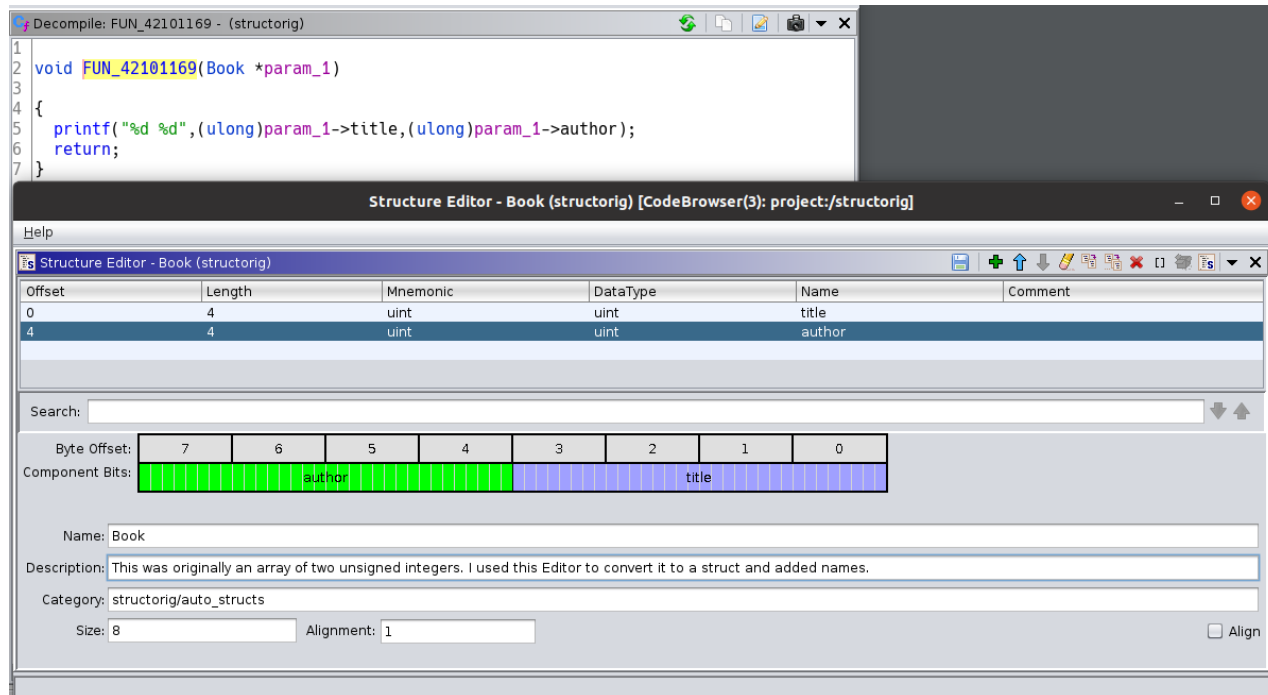
Parameters	Description
input0	First floating-point input to add.
input1	Second floating-point input to add.
output	Varnode containing result of addition.

#### Semantic statement

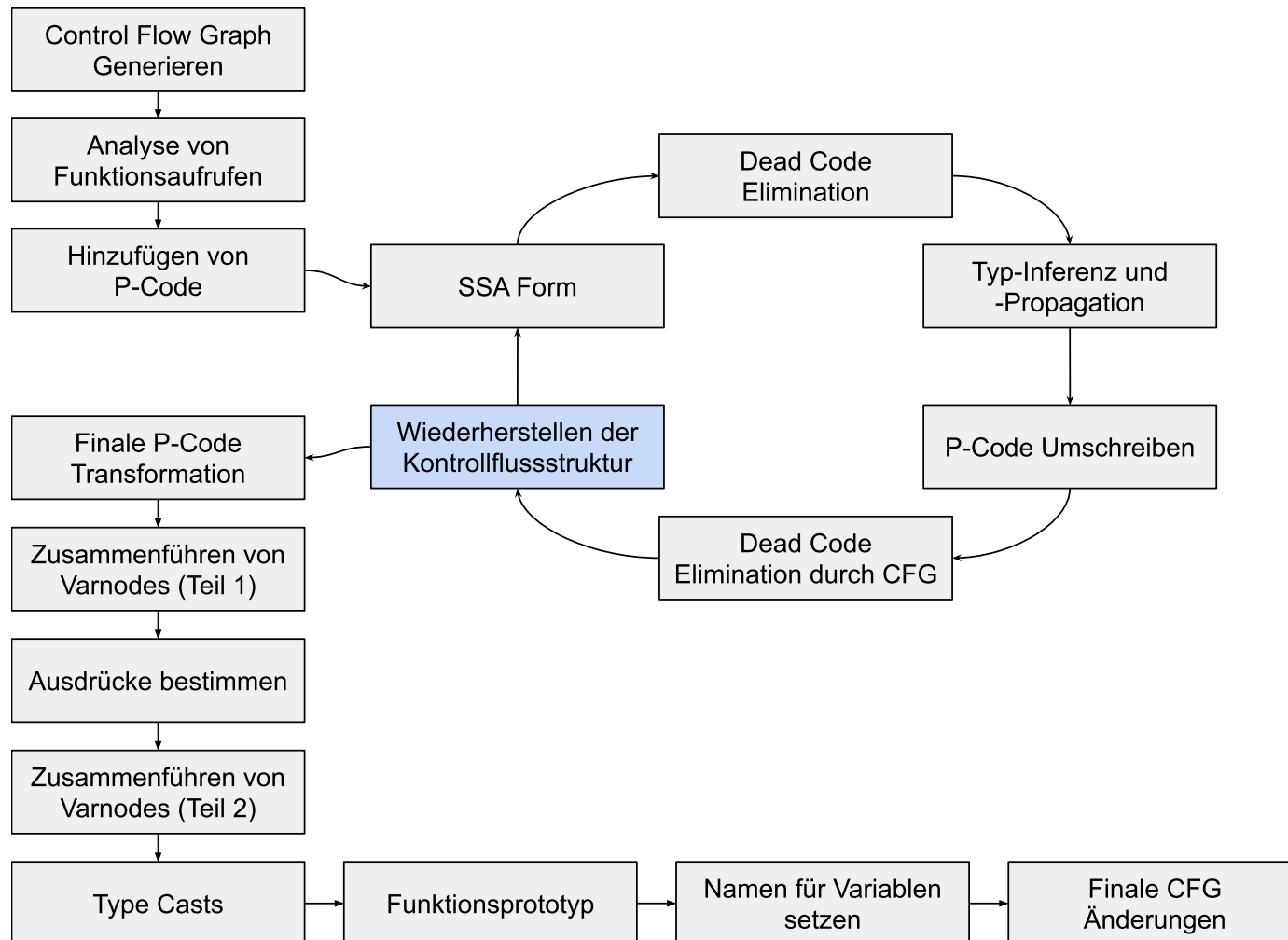
```
output = input0 f+ input1;
```

## 2.3 Typ-Inferenz und -Propagation

- Problematisch sind nicht primitive Datentypen
- Array, Struct, Class, Enum, ...



## 2.3 Kontrollflussstruktur herstellen

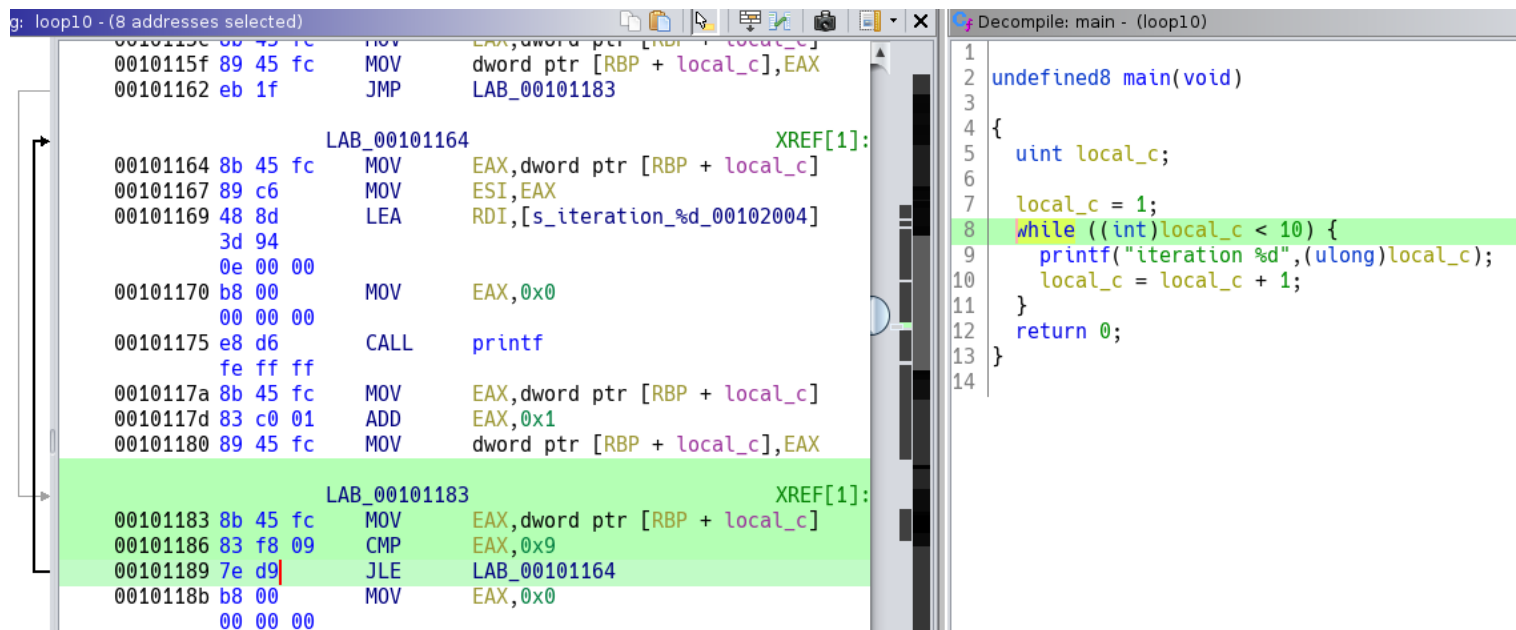


## 2.3 Kontrollflussstruktur herstellen

- Muster erkennen
- ADD-CMP-JLE Muster:

Source Code:

```
int main() {  
    for (int i = 1; i < 10; i++) {  
        printf("iteration %d", i);  
    }  
}
```



The screenshot displays a debugger window with two panes. The left pane shows assembly code for a loop, and the right pane shows the decompiled C code.

**Assembly View (Left Pane):**

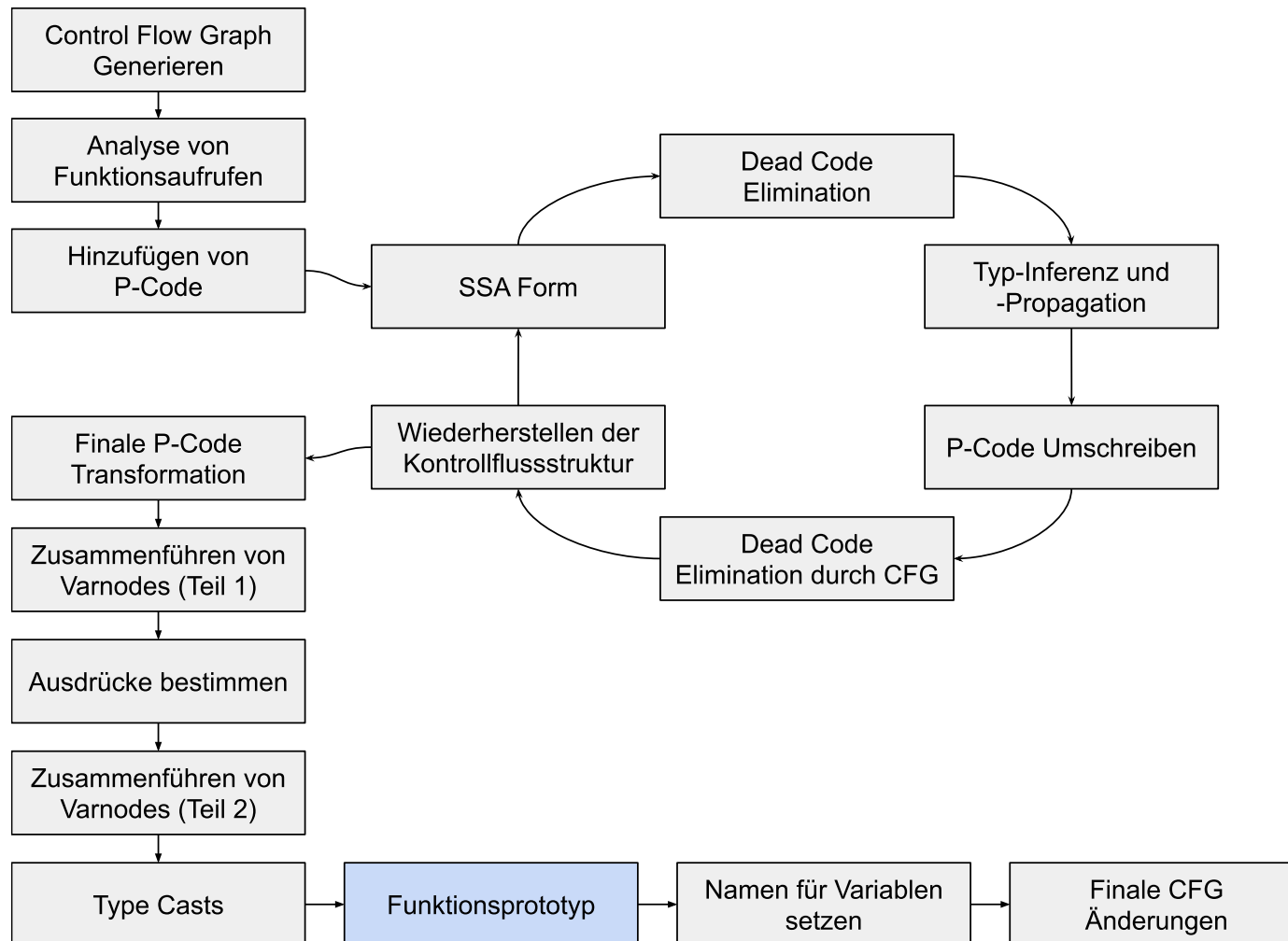
- Address 0010115c: `MOV EAX, dword ptr [RBP + local_c]`
- Address 0010115f: `MOV dword ptr [RBP + local_c], EAX`
- Address 00101162: `JMP LAB_00101183`
- LAB\_00101164:**
  - Address 00101164: `MOV EAX, dword ptr [RBP + local_c]`
  - Address 00101167: `MOV ESI, EAX`
  - Address 00101169: `LEA RDI, [s_iteration_%d_00102004]`
  - Address 00101170: `MOV EAX, 0x0`
  - Address 00101175: `CALL printf`
  - Address 0010117a: `MOV EAX, dword ptr [RBP + local_c]`
  - Address 0010117d: `ADD EAX, 0x1`
  - Address 00101180: `MOV dword ptr [RBP + local_c], EAX`
- LAB\_00101183:**
  - Address 00101183: `MOV EAX, dword ptr [RBP + local_c]`
  - Address 00101186: `CMP EAX, 0x9`
  - Address 00101189: `JLE LAB_00101164`
  - Address 0010118b: `MOV EAX, 0x0`

**Decompile View (Right Pane):**

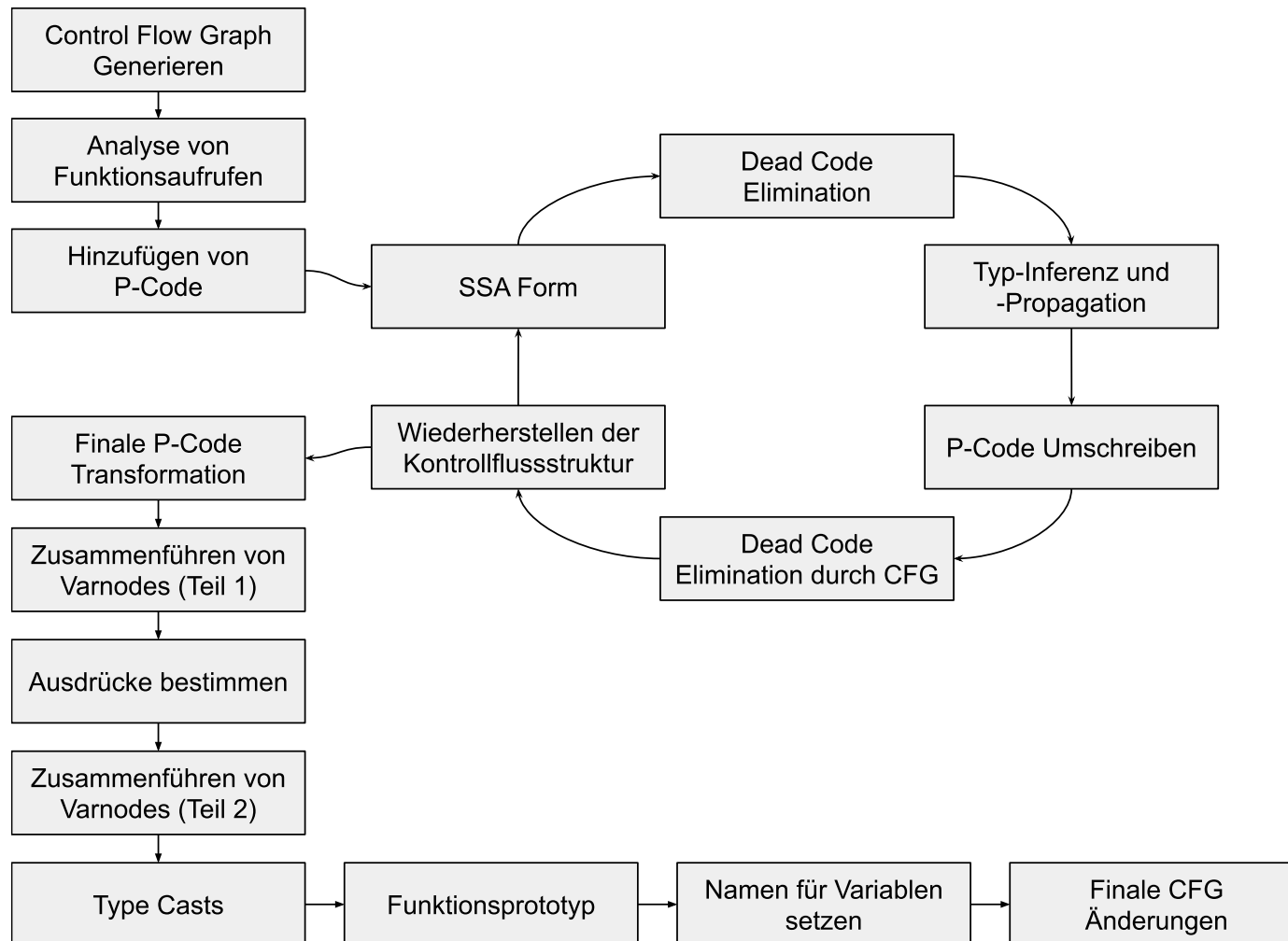
```
1 undefined8 main(void)  
2  
3 {  
4     uint local_c;  
5     local_c = 1;  
6     while ((int)local_c < 10) {  
7         printf("iteration %d", (ulong)local_c);  
8         local_c = local_c + 1;  
9     }  
10    return 0;  
11 }  
12  
13  
14
```



## 2.3 Funktionsprototyp

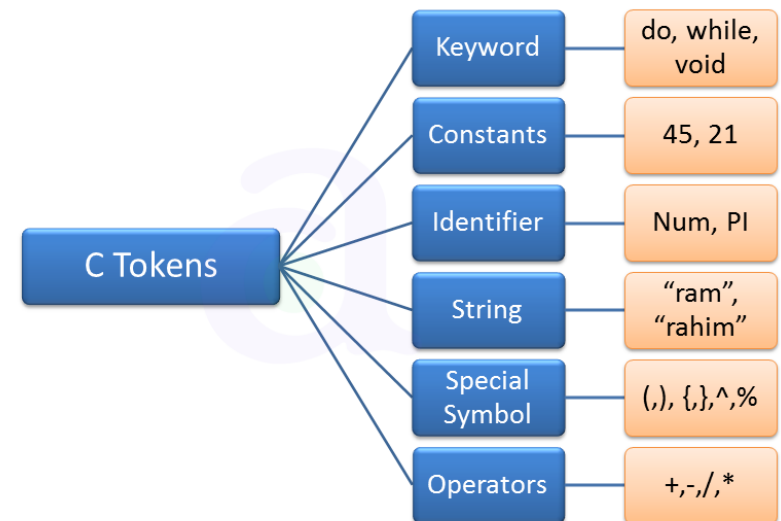


## 2.3 P-Code Optimierung



## 2.4 P-Code zu C Pseudocode

- P-Code und zusätzliche Informationen:  
Kontrollflussgraph, Funktionssignatur,  
Kontrollflussstrukturen, ...
- P-Code und Informationen  
durch C Tokens darstellen



[3]

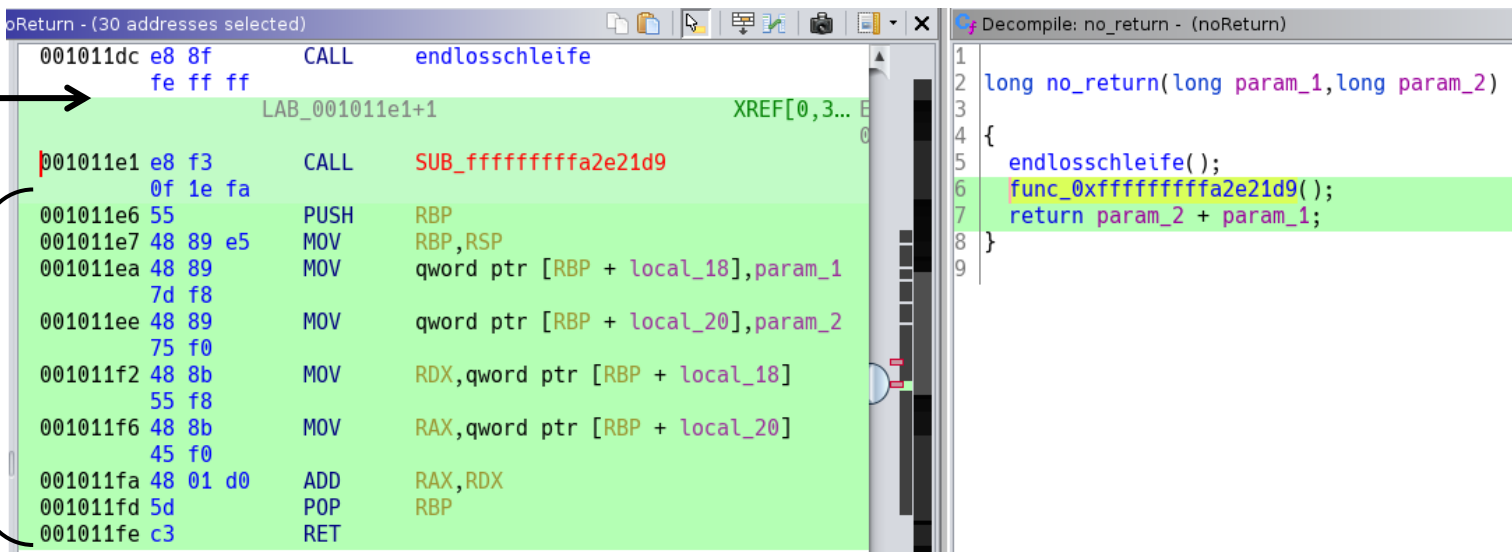
# 3.1 Non-returning Function

- Funktionsende nicht erkannt

```
void no_return() {  
    endlosschleife();  
}  
  
void endlosschleife () {  
    while(1) {}  
}
```

Kein 'return' →

Andere Funktion {

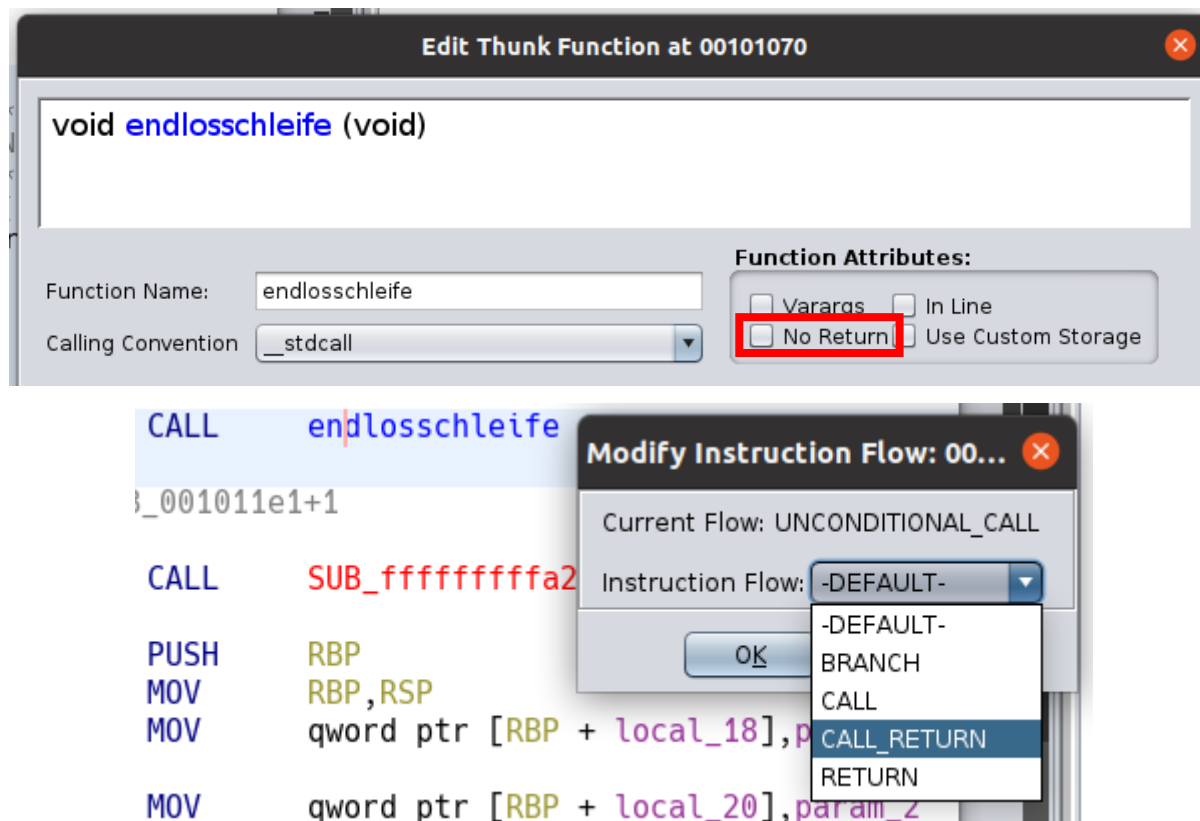


Address	Disassembly	Comment
001011dc	CALL	endlosschleife
001011e1	CALL	SUB_fffffffa2e21d9
001011e6	PUSH	RBP
001011e7	MOV	RBP, RSP
001011ea	MOV	qword ptr [RBP + local_18], param_1
001011ee	MOV	qword ptr [RBP + local_20], param_2
001011f2	MOV	RDX, qword ptr [RBP + local_18]
001011f6	MOV	RAX, qword ptr [RBP + local_20]
001011fa	ADD	RAX, RDX
001011fd	POP	RBP
001011fe	RET	

```
Decompile: no_return - (noReturn)  
1  
2 long no_return(long param_1, long param_2)  
3  
4 {  
5     endlosschleife();  
6     func_0xfffffffffa2e21d9();  
7     return param_2 + param_1;  
8 }  
9
```

## 3.1 Non-returning Function

- Funktion als ‚non-returning‘ markieren



## 3.2 Anti-Decompiler Techniken

- Code verschlüsselt oder komprimiert
- Erkennen durch Entropie Indikator
- Code zur Laufzeit aus Prozess auslesen

```

00101026 f2 ff    JMP      qword ptr [PTR_00103fc0]
          25 93
          2f 00 00
0010102d 0f      ??      0Fh
0010102e 1f      ??      1Fh
0010102f 00      ??      00h
00101030 f3 0f    ENDBR64
          1e fa
00101034 68 00    PUSH     0x0
          00 00 00
00101039 f2 e9    JMP      FUN_00101020
          e1 ff
          ff ff

-- Flow Override: CALL_RETURN (CALL_TERMINATOR)
0010103f 90      NOP
00101040 f3 0f    ENDBR64
          1e fa
    
```



## 3.3 Probleme in Decompiler Ausgabe

- P-Code in Ausgabe
- Variable/Register in Ausgabe startet mit:
  - ,in\_‘ oder ,unaff\_‘: Wert uninitialisiert

# Zusammenfassung Ghidra Decompiler

- Übersetzt Maschinencode zu C Pseudocode
- Programm Funktionsweise einfacher verstehen
- Kann nicht alle Informationen widerherstellen
- Kann Fehler machen
- Decompiler ist interaktiv



# Quellen

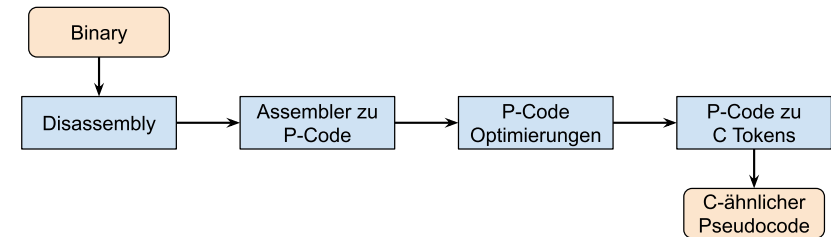
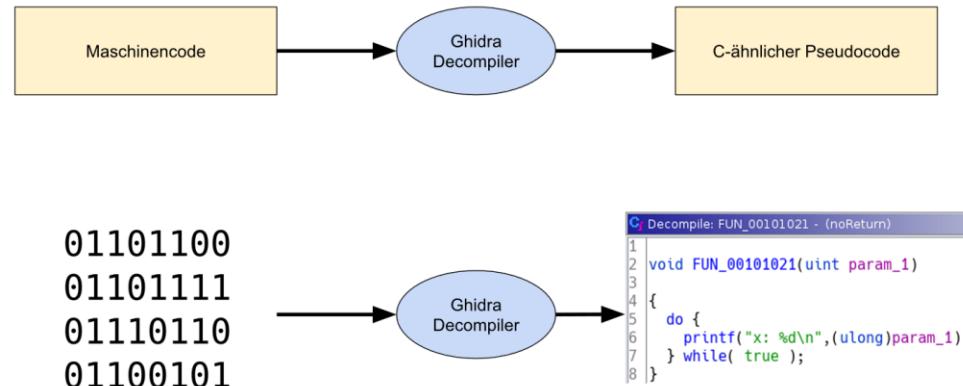
[1] <https://ghidra.re/courses/languages/html/pcoderef.html>

[2] <https://ghidra.re/courses/languages/html/pcodedescription.html>

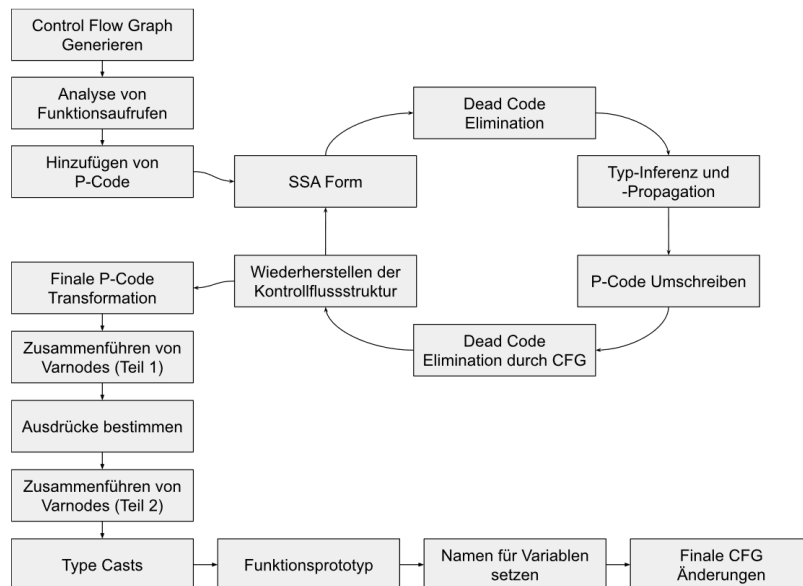
[3] <https://github.com/threatrack/ghidra-fidb-repo>

[4] <https://www.atnyla.com/tutorial/tokens-in-c/1/162>

# 1. Ghidra Decompiler



## 2.3 P-Code Optimierung



## 2.4 P-Code zu C Pseudocode

- P-Code und zusätzliche Informationen: Kontrollflussgraph, Funktionssignatur, Kontrollflussstrukturen, ...
- P-Code und Informationen durch C Tokens darstellen

