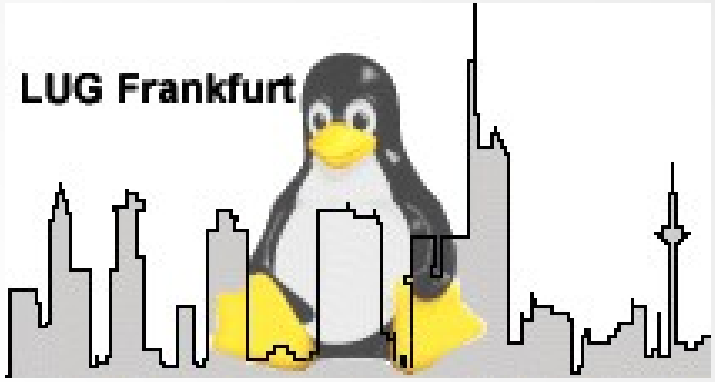


BIONIC

Wie sicher ist Androids Grundlage?

Dr. Christoph Zimmermann

Chemnitzer Linux Tage,10. 3. 2018

Für Luca

Inhalt

1. Übersicht
2. ISO 9126 Metriken
3. Angriffsoberflächenanalyse
4. Weitere Beobachtungen

stat /proc/self

- Promotion im Bereich reflektive Betriebssystemarchitekturen
- Start mit Linux: Kernel 0.95
- Tech Support @ FraLUG :-)
- Arch Package Maintainer
- Hobbies:
  - SDLC
  - IT Sicherheit und andere Formen schwarzer Kunst
  - Social Engineering mit Fokus auf kognitiver und Verhaltenspsychologie

Übersicht

- Was ist Bionic
  - Android Laufzeitumgebung (vergleichbar mit libc in Standard Linux Systemen)
  - Verbindung zwischen Kern und Anwendungen (inkl. Java VMs)
- Warum ist dies wichtig?
  - Grundlage für alle Applikationen – jede Sicherheitslücke hast Auswirkungen auf das User-Land
- Welche Implikationen?
  - Angriffsoberflächenanalyse
  - Und Behebung

Android

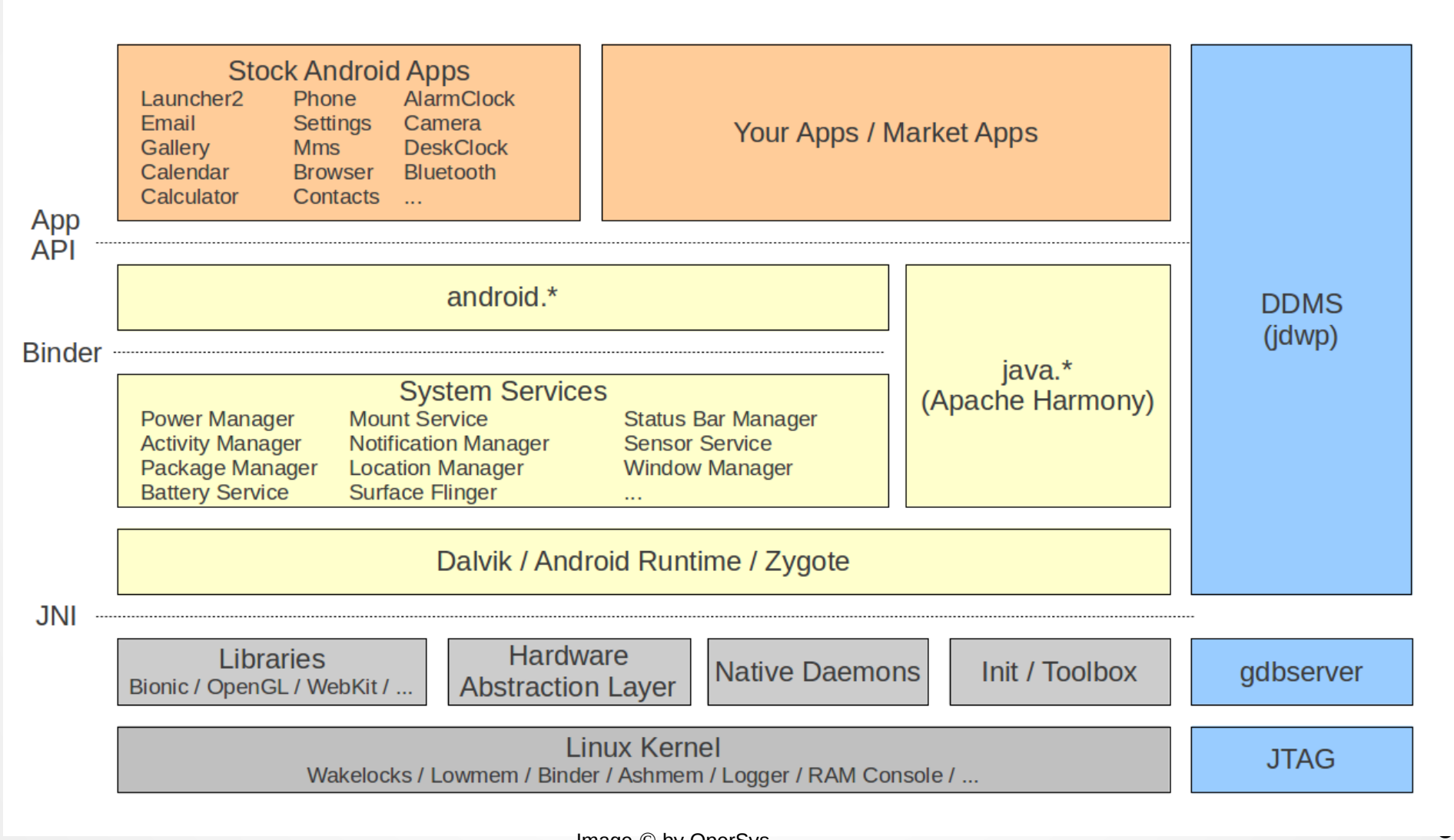


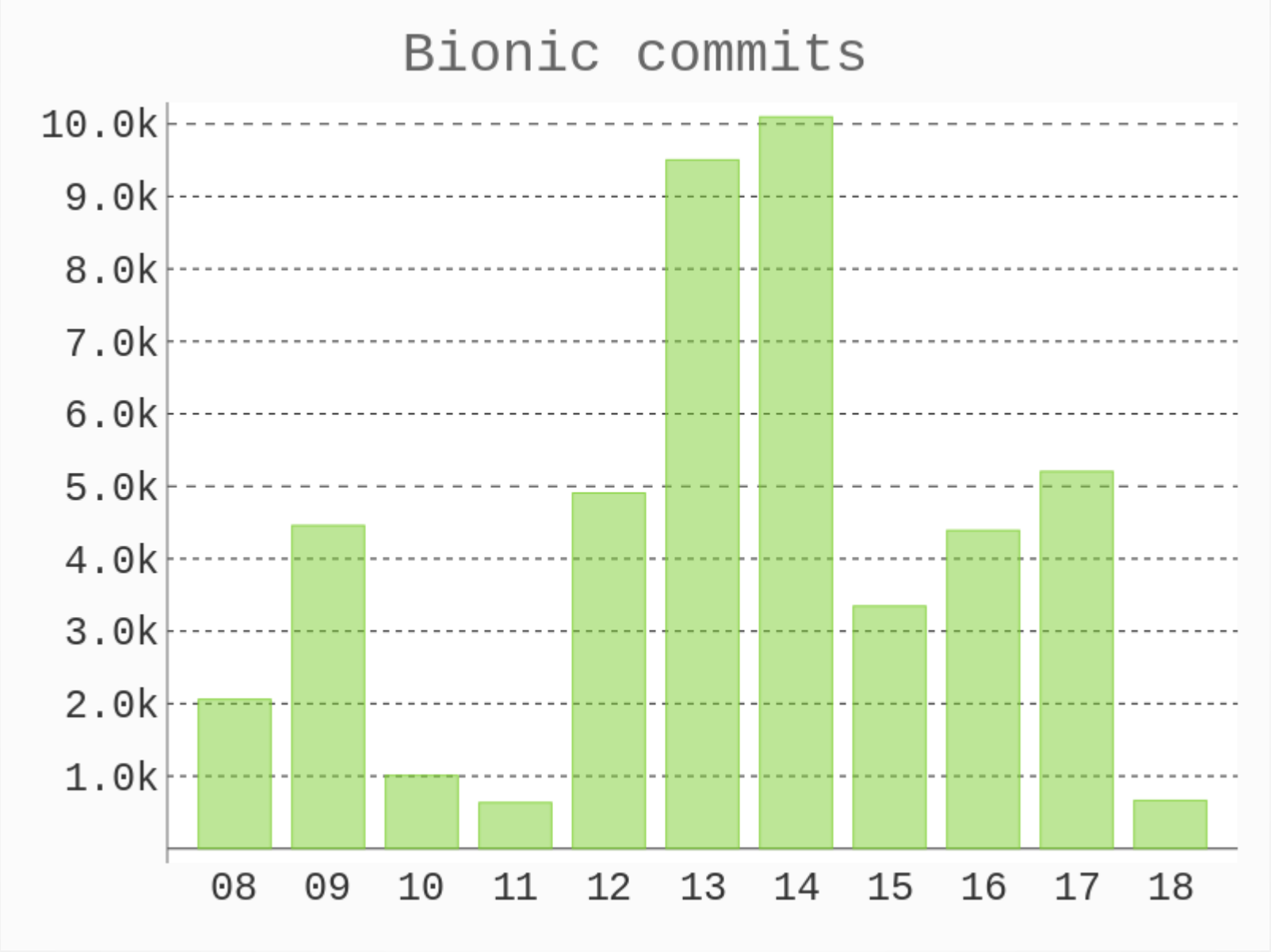
Image © by OperSys

Umfang

- Analyse der Bionic Code-Basis:
  - Im Hinblick auf Wartbarkeit (ISO 9126)
  - Identifikation der generellen Angriffsoberfläche
  - Weitere Resultate aufgrund von zusätzlicher Analyse
  - Übersicht über H/L Mitigation

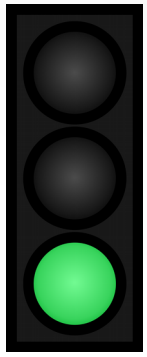
Bewertung

- Werkzeuge:
  - SonarQube
  - RATS (Rough Auditing Tool for Security)
  - Cppcheck
  - Gesunder Menschenverstand + 30 Jahre SDLC-Erfahrung
  - Andere Formen schwarzer Magie ☺
- Code-Basis:
  - android.googlesource.com/platform/bionic.git


Bionic commits
10.0k
9.0k
8.0k
7.0k
6.0k
5.0k
4.0k
3.0k
2.0k
1.0k
08
09
10
11
12
13
14
15
16
17
18

Summary

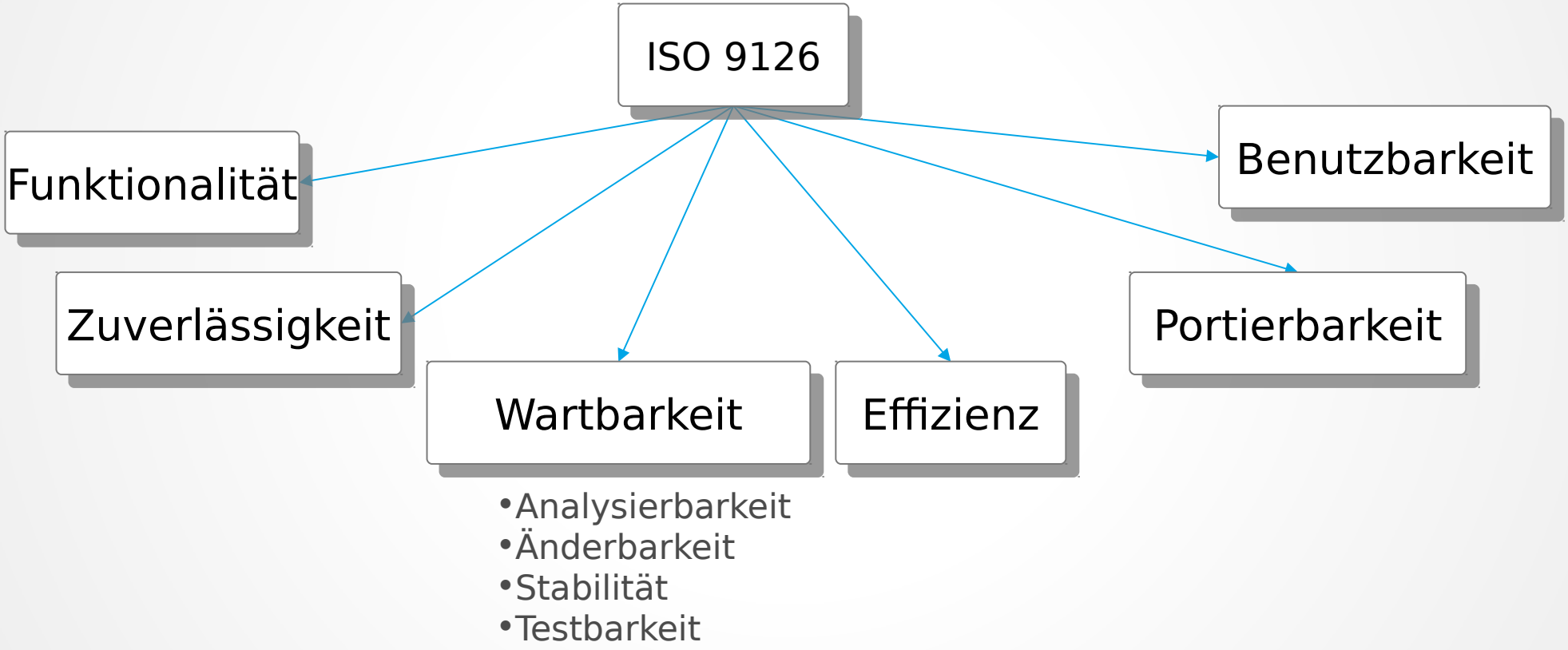
- Gesamtqualität:



- Aber:
  - Einige Sicherheitsrisiken durch unsichere Codierungspraktiken
  - Viele Code Smells
  - Intensive Benutzung von Uralt-Code

ISO 9126

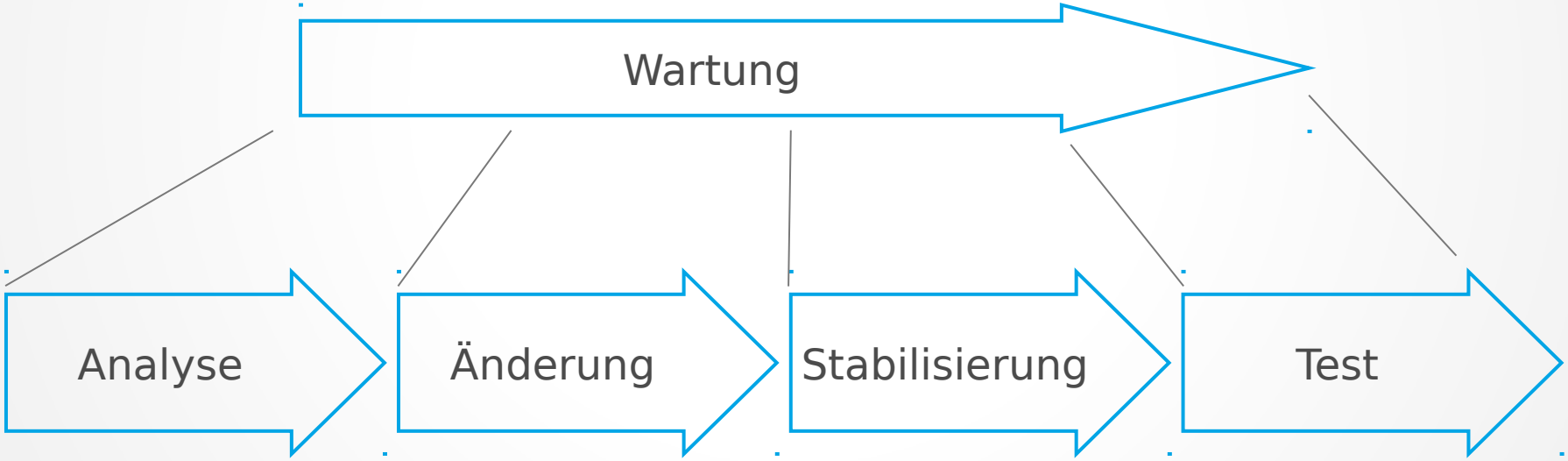
- Internationaler Standard für S/W-Qualitätsanalyse



Wartbarkeitsattribute

Wartbarkeit =

- *Analysierbakeit*: Wie und wo zu ändern?
- *Änderbarkeit*: Wie einfach ist eine Änderung?
- *Stabilität*: Code-Kohärenz wärend der Änderung?
- *Testbarkeit*: Validierung der Änderung?



Vereinfachtes Analyse-Model

	Volume	Duplizierung	Einheit-Komplexität
Analysierbarkeit	X	X	
Änderbarkeit		X	X
Stabilität	X	X	X
Testbarkeit			X

Volumen

- Software Produktivität:
  - xLOC
  - Function points (FPs)
  - …
- Herausforderung:
  - Ausdruckskraft von unterschiedlichen Programmiersprachen
  - Ansatz: Gewichtung xLOC mit Standard-Produktivitätsfaktor
  - ➢ Programming Languages Table

Volumen (ff.)

- Programming Languages Table:

Sprache	Level	Durchschn. # LOC pro FP
Perl	15	21
Smalltalk/V	15	21
Objective C	12	27
Haskell	8.5	38
C++	6	53
Basic	3	107
C	2.5	128
Macro assembler	1.5	213

Volumen (ff.)

- Warum ist das wichtig:
  - Gesamtkosten
  - Aufwand für Neuerstellung
  - Bionic Volumenmetriken:

Einheit	#
Gesamt-LOCs	422,969
Dateien	3,981
Funktionen	5,597
Klassen	336

Duplizierung

- Code-Duplizierung reduziert Wartbarkeit
  - Hohe Wartungskosten
  - Fehlerbehebung
  - Reduzierte Testbarkeit

Duplizierung (ff.)

- Bionic Duplizierungsmetriken:

Einheit	Duplizierung
Gesamt	0.9%
Blöcke	127
Dateien	49

Einheits-Komplexität

- Wird gemessen mittels McCabe's zyklischer Komplexität:
  - Anzahl von Entscheidungspunkten (decision points / DPs) pro Einheit (Methode / Funktion / Datei)
  - McCabe, IEEE Transactions on Software Engineering, 1976
  - Höhere Komplexität bedingt höherer Aufwand bei Änderungen und Test
  - Für C/C++/Objective C, Erhöhung von DPs für:

Funktionsdefinitionen, while, do while, for, throw statements, return (Ausnahme: letzte Anweisung einer Funktion), switch, case, default, &&, ||, ?, catch, break, continue, goto

Einheits-Komplexität (ff.)

- Übersicht:

Zyklische Komplexität	Risiko-Einschätzung
1 - 10	Klarer Code, geringes Risiko
11 - 20	Komplex, mittleres Risiko
21 - 50	Sehr komplex, hohes Risiko
> 50	Nicht verständlich, sehr hohes Risiko

Einheits-Komplexität (ff.)

- Bionic Komplexitätsmetrik:

Einheit	Komplexität
Funktion	3.4
Klasse	0.2
Datei	5.7

Zusammenfassung

- Ergebnis der Code-Analyse: sehr gut
  - SQALE Rating: A
  - Geschätzte technische Schuld: 17 Tage
- Aber einige Sicherheitsprobleme:

Einheit	Vorkommen
Schwachstelle	1
Kleinere Probleme	74
Smells	1,634

(SQALE: Software Quality Assessment based on Lifecycle Expectations)

Angriffsoberflächenanalyse

- Ergebnis:
  - Kein Refactoring nötig
  - Angriffsoberflächenanalyse:
    - Nur eine Schwachstelle
  - Kleinere Probleme:
    - Time of check / time of use Probleme
    - Potentielle Memory Leaks
    - Klasseninitialisierungen

Angriffsoberflächenanalyse (ff.)

- Angriffsoberflächenanalyse (ff.):
  - Smells: Überwiegend String und Puffer-Probleme
  - Primär durch exzessive Wiederverwendung von Uralt-Code
- Abhilfe:
  - Erweiterte Code-Review
  - Einsatz von statischen Code-Analyse Werkzeugen
  - Behebung von Code-Problemen

Angriffsoberflächenanalyse (ff.)

- linker.cpp (#351): CWE-562, return of stack variable address

```
static bool realpath_fd(int fd, std::string* realpath) {
  std::vector<char> buf(PATH_MAX), proc_self_fd(PATH_MAX);

  __libc_format_buffer(&proc_self_fd[0], proc_self_fd.size(),
"/proc/self/fd/%d", fd);

  if (readlink(&proc_self_fd[0], &buf[0], buf.size()) == -1) {
    PRINT("readlink(\"%s\") failed: %s [fd=%d]", &proc_self_fd[0],
strerror(errno), fd);
    return false;
  }

  *realpath = &buf[0];
  return true;
}
```

Angriffsoberflächenanalyse (ff.)

- Typische smells:
  - libc/arch-mips/string/memcpy.c: Keine Überprüfung des Parameters len

```
memcpy (void *a, const void *b, size_t len) __overloadable
```

  - libc/arch-arm/bionic/atexit_legacy.c: variabler Format-String

```
static char const warning[] = "WARNING: generic atexit() called
from legacy shared library\n";

   __libc_format_log(ANDROID_LOG_WARN, "libc", warning);

    fprintf(stderr, warning);
```

Generelle Behebung

1. Reduktion der Angriffsoberfläche durch Eliminierung der Sicherheitsrisiken (s. v. Folie)
2. Reduktion der Komplexität von ausgesuchten Modulen
3. Reduktion der geringen Duplizierung durch Restrukturierung des entsprechendes Quellkodes
4. Identifikation von großvolumigen Einheiten und Restrukturierung der Code-Basis

Zusammenfassung

- Solide Code-Basis trotz Alter
- Minimale Angriffsoberfläche: kein generelles Refactoring notwendig
- Kleinere Probleme können ohne großen Aufwand behoben werden
- Komplexes Angriffsszenario für Schwachstellen-ausnutzung

➢ Stabile Code-Basis für das verbleibende User-Land

Quellen

- Bionic source code: android.googlesource.com/platform/bionic.git
- Sonarqube: www.sonarqube.org/downloads
- Cppcheck: cppcheck.sourceforge.net
- RATS: code.google.com/archive/p/rough-auditing-tool-for-security/downloads

Diskussion / Fragen

Vielen Dank!

Dr. Christoph Zimmermann

monochromec at <ignore>space</ignore>gmail<dot></dot>com