# Finding Use-After-Free vulnerabilities using Symbolic Execution with angr

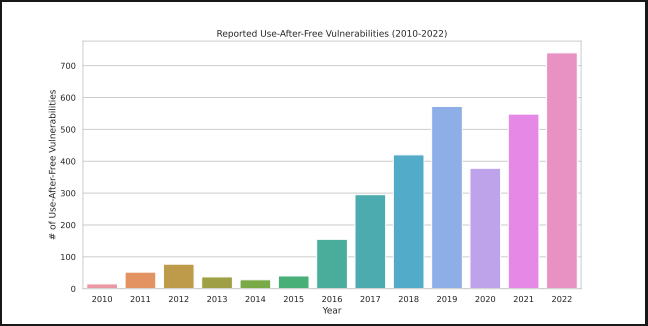
## Abschlussvortrag Bachelorarbeit

Lennart Henke

Rheinische Friedrich-Wilhelms-Universität Bonn

# Motivation

## Use-After-Free Schwachstellen




Anstieg von gemeldeten Use-After-Free Schwachstellen in der NVD

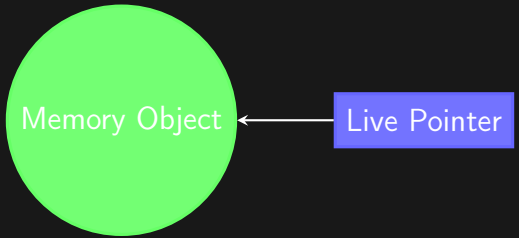
# Motivation

## Warum Symbolic Execution?


- Verwendung statischer Analyse zur Erkennung von Schwachstellen
- Problem: False Positives erfordern manuelle Überprüfung
- Lösung: Verifikation mit Symbolic Execution

# Use-After-Free

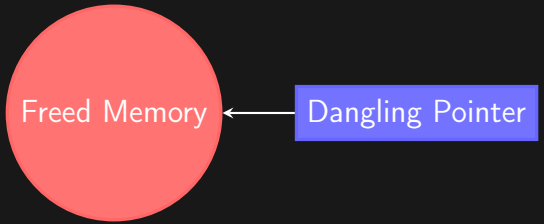
## Schwachstelle




1. Speicher wird allokiert, wenn eine Anwendung dynamisch Speicherplatz reserviert.

# Use-After-Free

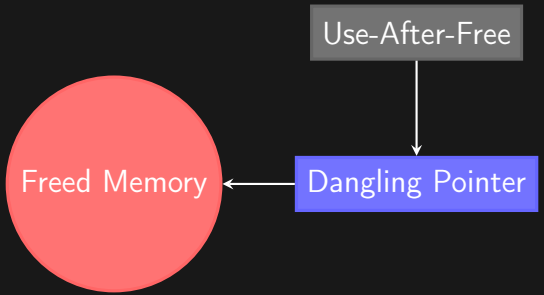
## Schwachstelle




2. Dangling Pointer entsteht, wenn ein Live-Pointer auf ein freigegebenes Speicherobjekt zeigt.

# Use-After-Free

## Schwachstelle




3. Use-After-Free tritt auf, wenn ein Dangling Pointer verwendet wird, um auf freigegebenen Speicher zuzugreifen.

# Use-After-Free

Einfaches Beispiel

```
1  void example_func(int a, int b) {
2      int* ptr = (int*) malloc(sizeof(int)); // allocation
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7              x = 2 * (a + b);
8      }
9      free(ptr);  // deallocation
10     if (x - y == 0)
11         *ptr = 42; // use-after-free
12 }
```

# Use-After-Free

Einfaches Beispiel

```
1  void example_func(int a, int b) {
2      int* ptr = (int*) malloc(sizeof(int)); // allocation
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7              x = 2 * (a + b);
8      }
9      free(ptr);  // deallocation
10     if (x - y == 0)
11         *ptr = 42; // use-after-free
12 }
```

# Use-After-Free

Einfaches Beispiel

```
1  void example_func(int a, int b) {
2      int* ptr = (int*) malloc(sizeof(int)); // allocation
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7              x = 2 * (a + b);
8      }
9      free(ptr);  // deallocation
10     if (x - y == 0)
11         *ptr = 42; // use-after-free
12 }
```

# Use-After-Free

Einfaches Beispiel

```
1  void example_func(int a, int b) {
2      int* ptr = (int*) malloc(sizeof(int)); // allocation
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7              x = 2 * (a + b);
8      }
9      free(ptr);  // deallocation
10     if (x - y == 0)
11         *ptr = 42; // use-after-free
12 }
```

# Symbolic Execution

## Theorie

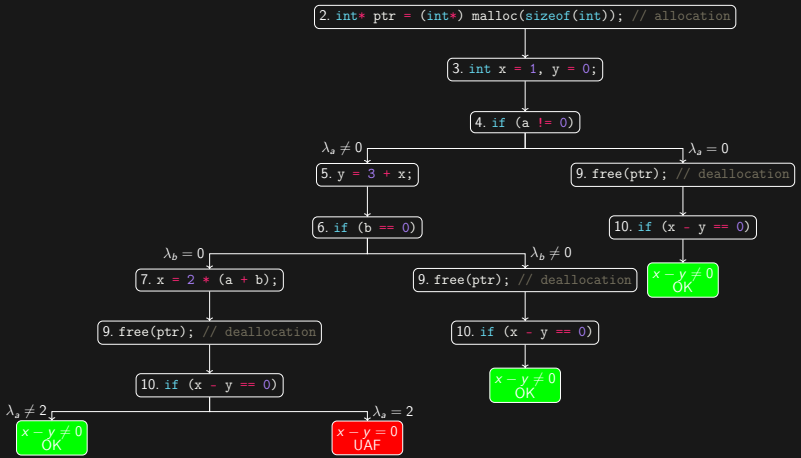
Symbolic Execution ermöglicht die Ausführung eines Programms mit symbolischen Eingabewerten anstelle von konkreten Werten.

Es funktioniert folgendermaßen:

1. Initialisierung: Einfügen von Symbolen
2. Pfad-Erkundung: Verzweigen für jeden Ausführungspfad
3. Constraints-Solving: Lösen von Pfadbeschränkungen

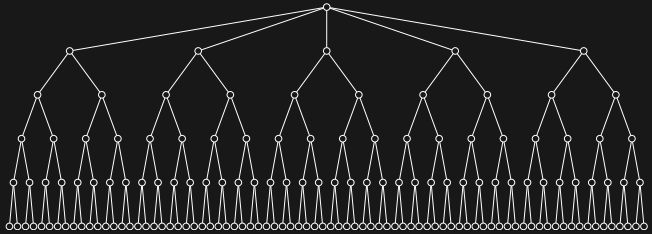
# Symbolic Execution

## Einfaches Beispiel



# Symbolic Execution

## Pfadexplosion Problem



- Theoretisch ist Symbolic Execution ein korrekter und vollständiger Ansatz für jede entscheidbare Analyse
- Problem: Die Anzahl der Pfade steigt exponentiell mit der Anzahl der Programmverzweigungen

# angr

## Übersicht

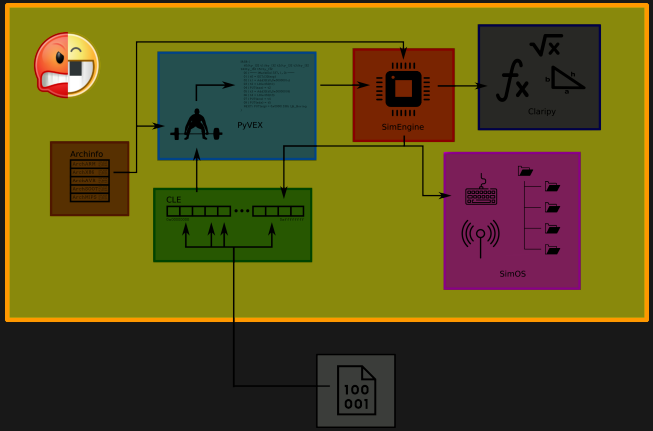
Binäranalyse-Framework, das Symbolic Execution unterstützt

Funktionen:

- Schrittweise durch Binärdateien gehen (und jeder Verzweigung folgen)
- nach einem Programmzustand suchen, der ein bestimmtes Kriterium erfüllt
- Lösen von symbolischen Variablen bei gegebenen Pfadbeschränkungen

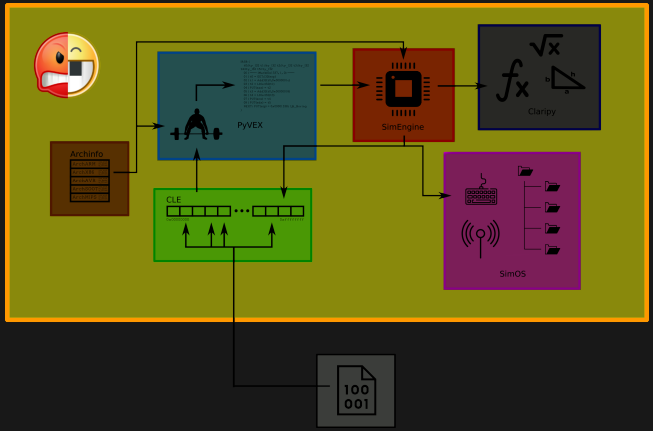
# angr

## Komponenten



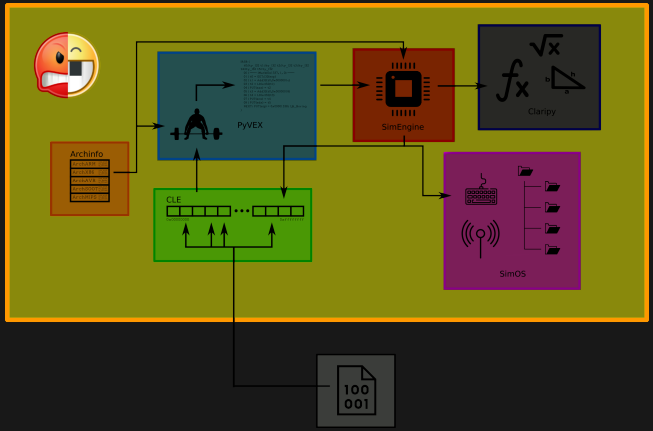
# angr

## Komponenten



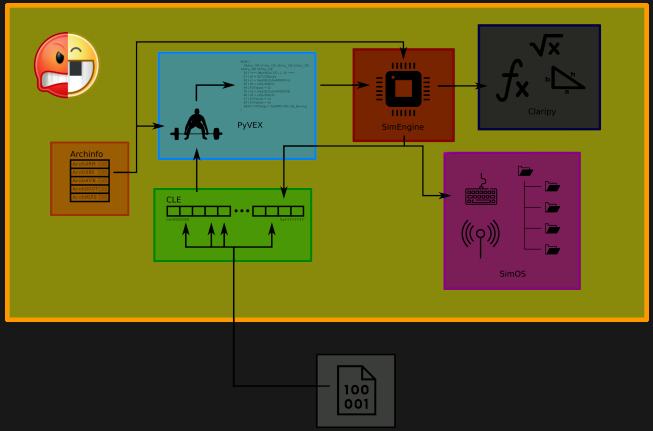
# angr

## Komponenten



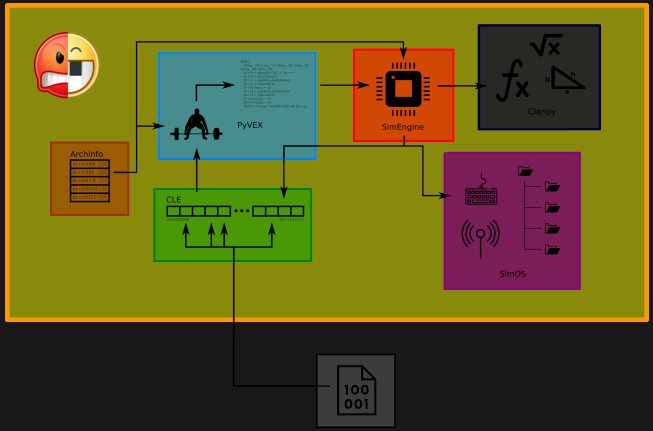
# angr

## Komponenten



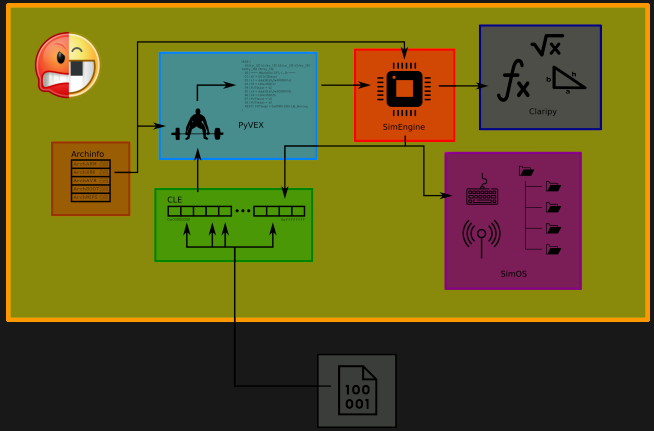
# angr

## Komponenten



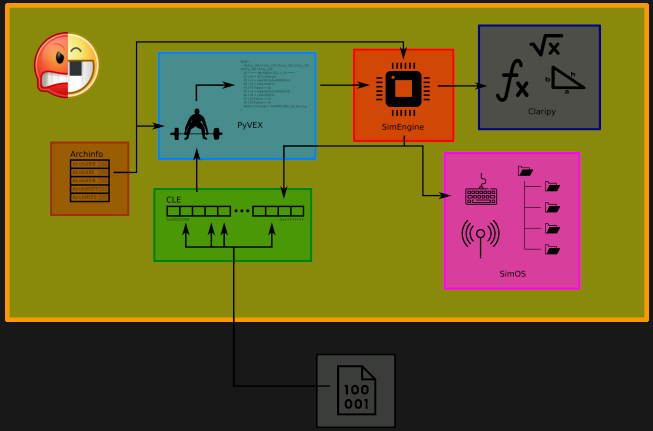
# angr

## Komponenten



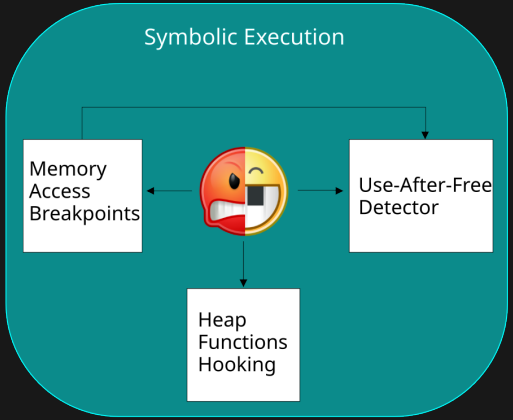
# angr

## Komponenten



# Use-After-Free Analyse

## Schwachstellen Erkennung



# Use-After-Free Analyse

## Überwachung von Speicherzugriffen

| Pointer | Größe | Allocation | Free |
|---|---|---|---|
| <BV64 0xc0000f40> | <BV64 0x4> | 0x401211 | 0x401317 |
| <BV64 0xc0000f50> | <BV64 size_1_64> | 0x401264 | None |
| <BV64 0xc0000fd0> | <BV64 0x8> | 0x4012b9 | 0x40132f |
| UNINITIALIZED | <BV64 size_2_64> | <BV64 malloc_site_64> | 0x401323 |

- Überwachen von Speicherregionen während der Programmausführung
- Claripy prüft, ob eine Zuweisung vorhanden ist, bei der ein Speicherzugriff innerhalb einer freigegebenen Region liegt

# Use-After-Free Analyse

## Programmausgabe

```
[+] Use-After-Free: <SimState @ 0x401189> in main may access a dangling pointer
    Call stack: __libc_start_main -> main
    Allocation-site: @ 0x401159
    Free-site:       @ 0x40116b
    Use-site:        @ 0x401189
    Free-ptr:        @ <BV64 0xc0000f40>
    Use-ptr:         @ <BV64 0xc0000f40>
    Size:              <BV64 0x4>
    Registers:
        rax: <BV64 0xc0000f40>
        rcx: <BV64 0x0>
        rdx: <BV64 0x7fffffffff0010 + 0x8 * mem_7fffffffff000...
        rbx: <BV64 reg_28_8_64{UNINITIALIZED}>
        (truncated...)
    Path:
        1: 0x401060 - _start
        2: 0x500000 - __libc_start_main
        (truncated...)
        10: 0x401189 - main
```

# Reduzierung von Pfadexplosion

Speicherverbrauch, Library Calls, Schleifen

- DFS anstelle von BFS
- AutoDrop zur automatischen Verwerfung irrelevanter Zustände
- MemoryWatcher zur Beendigung des Programms bei zu hohem Speicherverbrauch
- SimProcedures zur Simulation von Library Calls
- LoopSeer für die Begrenzung von symbolischen Schleifen

# Veritesting

## State Merging

- Zusammenführen mehrerer Pfade zu einem einzigen Pfad mithilfe einer disjunktiven Normalform des Programmzustands
- Wechselt zwischen DSE und SSE
- Zusammenführen kann komplexe Ausdrücke einführen, die den Constraint Solver überlasten können

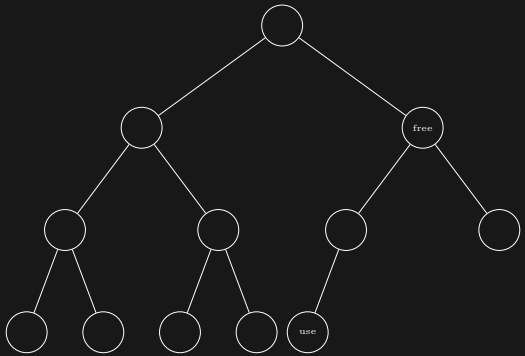
# Under-Constrained Symbolic Execution

## Zielgerichtete Analyse

- Direktes Ausführen einzelner Funktionen innerhalb eines Programms
- Weniger Code = weniger Pfade
- Herausforderungen:
  - Unzureichende Einschränkungen: Eingaben haben unbekannte Vorbedingungen
  - Welche Funktionen sollen analysiert werden?
  - Welche Funktionsaufrufe sollen ignoriert werden?

# Directed Symbolic Execution

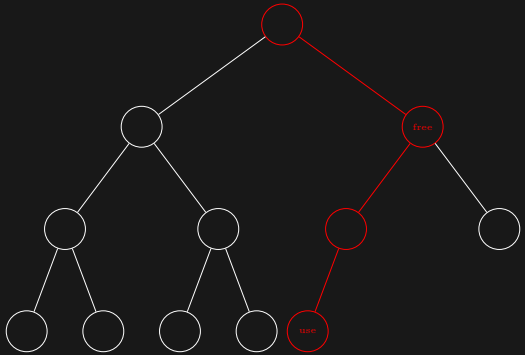
## Path-Pruning durch statische Analyse




- Priorisierung gefährdeter Pfade und Verwerfung uninteressanter Pfade

# Directed Symbolic Execution

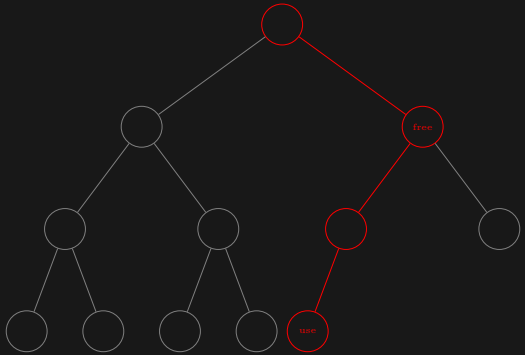
## Path-Pruning durch statische Analyse




- Priorisierung gefährdeter Pfade und Verwerfung uninteressanter Pfade

# Directed Symbolic Execution

## Path-Pruning durch statische Analyse




- Priorisierung gefährdeter Pfade und Verwerfung uninteressanter Pfade

# Evaluation

## Setup

- Juliet Test Suite 1.3 für C/C++ zur Genauigkeitsbewertung der UAF-Analyse
- Bewertung der Performance in der Praxis durch Untersuchung realer Binärdateien, die UAF-CVEs enthalten
- Maximale Pfadlänge von 3000
- Timeout von 3 Stunden
- Verwendet BFS und beschränkt die Speichernutzung auf 120 GB zur Laufzeitverkürzung

# Evaluation

## Juliet Test Suite

| CWE-416: Use-After-Free | |
|---|---|
| Positive cases: | 393 |
| Negative cases: | 393 |
| True Positives: | 393 |
| False Positives: | 0 |
| True Negatives: | 393 |
| False Negatives: | 0 |
| True Positive Rate: | 100% |
| True Negative Rate: | 100% |
| Accuracy: | 100% |

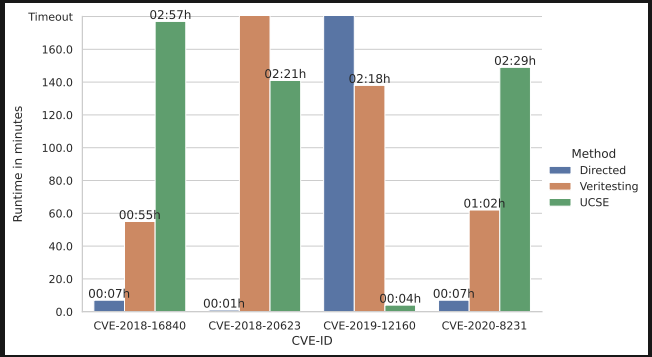
# Evaluation

## Reale Binärdateien

| Known UAFs | | Number of reported UAFs | | | |
|---|---|---|---|---|---|
| Identifier | Program | cwe_checker | Directed | UCSE | Veritesting |
| CVE-2019-12160 | GoHttp | 1 | 0 | 0 | 0 |
| CVE-2018-20623 | readelf-2.31.1 | 7 | 0 | 0 | - |
| CVE-2018-16840 | curl-7.61.1 | 103 | 0 | - | 0 |
| CVE-2020-8231 | curl-7.71.1 | 105 | 0 | - | 0 |
| Total | | 216 | 0 | 0 | 0 |

# Evaluation

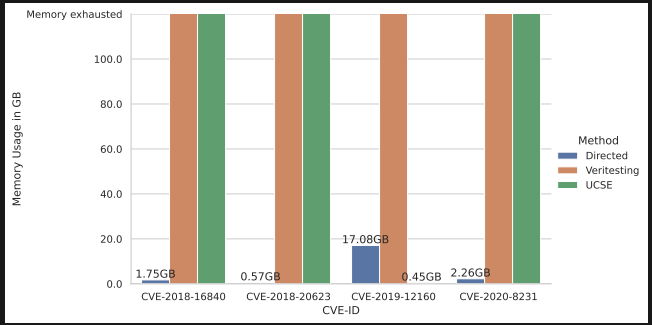
## Vergleich der Pfadexplosion-Minderungsmethoden




Laufzeit

# Evaluation

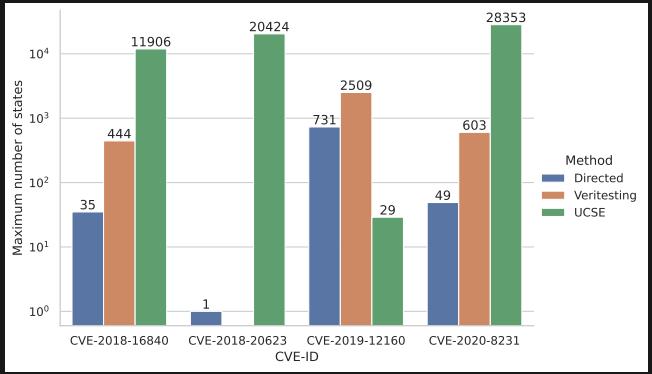
## Vergleich der Pfadexplosion-Minderungsmethoden




Speicherverbrauch

# Evaluation

## Vergleich der Pfadexplosion-Minderungsmethoden




Maximale Anzahl von aktiven Zuständen

# Evaluation

## Einschränkungen

- Pfadexplosion beeinträchtigt weiterhin die Skalierbarkeit
- SimProcedures simulieren nicht immer alle Bibliotheksfunktionen genau
- Speicher-Modell unterstützt nur begrenzte symbolische Lese- und Schreibvorgänge
- angr's Heap-Implementierung konkretisiert alle symbolischen Allokationsgrößen

# Evaluation

## Zukünftige Arbeit

- Engere Integration der Ergebnisse der statischen Analyse
- UCSE bei Funktionen beginnen, die "Dangling Pointer“ erzeugen
- Entwicklung einer speichereffizienteren Veritesting-Methode
- Behandlung unendlicher konkreter Schleifen

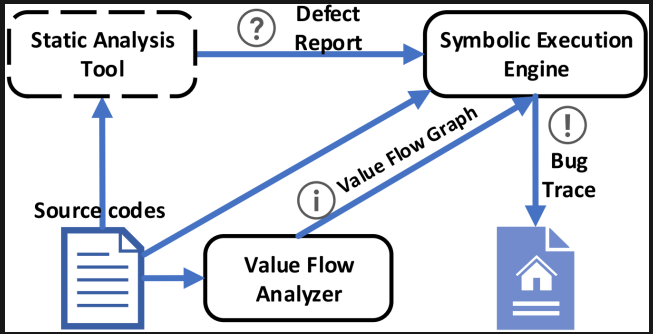
# Verwandte Arbeiten

Use-After-Free Erkennung

| Kategorie | Name | Mittel |
|---|---|---|
| Statische Source Code Analyse | Tac | Pointer-Alias-Analyse mit maschinellem Lernen |
| | CRed | Spatial-temporal Kontextreduktion |
| Statische Binärcode Analyse | UAFDetector | Funktions-Zusammenfassungen |
| Dynamische Analyse | Undangle | Taint-Analyse und Pointer-Verfolgung |

# Verwandte Arbeiten

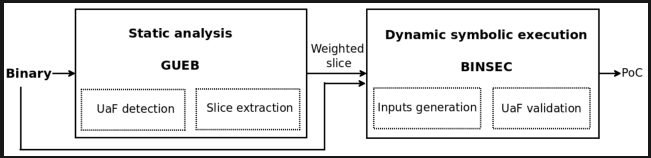
SATracer




- Nutz LLVM $\rightarrow$ benötigt Source Code
- Reduzierung von 71,4% der Fehlalarme von ClangSA
- fünf False Negaitves für die Juliet Test Suite

# Verwandte Arbeiten

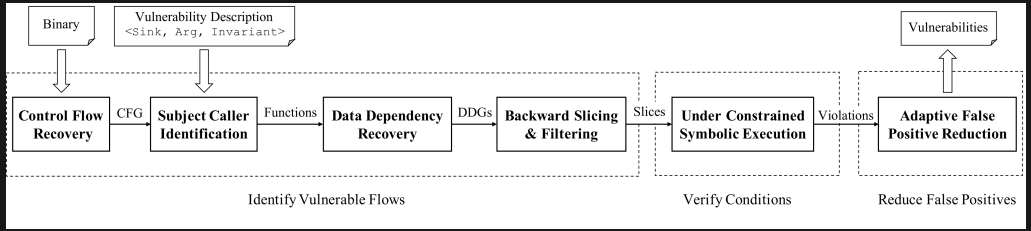
## Finding the Needle in the Heap




- Verifikation mittels Symbolic Execution, die stark an die statische Analyse gebunden ist
- Double Free in JasPer - CVE-2015-5221

# Verwandte Arbeiten

## Arbiter




- CWE-131, CWE-252, CWE-134 und CWE-337
- Nutzt angr und UCSE

# Zusammenfassung

tl;dr


- 100% Genauigkeit bei der Juliet Test Suite
- Findet keine der echten Schwachstellen
- Future Work: Stärkere Integration der statischen Analyse

Vielen Dank für eure Aufmerksamkeit!

# Bilder

https://angr.io/img/angry_face.png

http://lujie.ac.cn/files/papers/SATRACER.pdf

http://www.robindavid.fr/publications/ssprew-2016.pdf

https://raw.githubusercontent.com/jkrshnmenon/arbiter/master/overview.png

https://web.archive.org/web/20190908022725im_/http://deniable.org/imgs/z3/sat1.png

# Use-After-Free Analyse

## Detektor Algorithmus

**Require:** state, use-addr
1 **foreach** *memory block in state's heap* **do**
2  **if** *memory block is marked as free* **then**
3  in-range $\leftarrow$ (block-start $\leq$ use-addr) $\wedge$ (use-addr $<$ block-end);
4  **if** *the size of the memory block is symbolic* **then**
5  set min and max constraints on the size;
6  **end**
7  **if** *use-addr is symbolic and not uninitialized* **then**
8  set heap bounds constraints on the use-addr;
9  **end**
10  **if** *use-addr is uninitialized or block-start is uninitialized* **then**
11  use-after-free $\leftarrow$ (use-addr == block-start);
12  **else**
13  use-after-free $\leftarrow$ state's solver is able to satisfy in-range;
14  **end**
15  **if** *use-after-free* **then**
16  save a copy of the state;
17  **end**
18  **end**
19 **end**