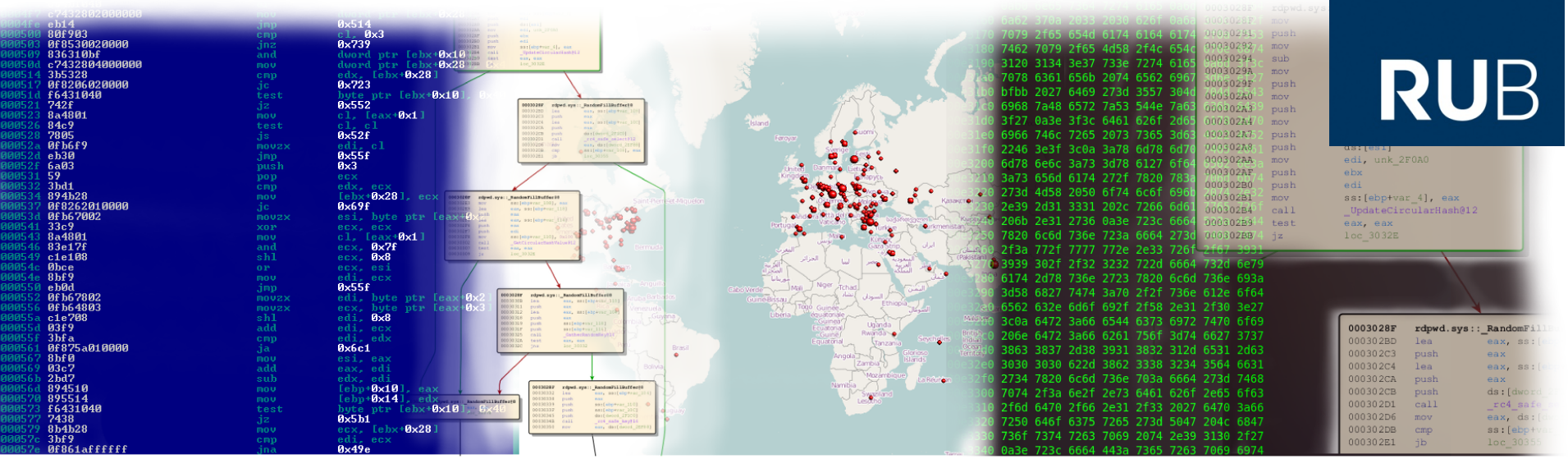


RUHR-UNIVERSITÄT BOCHUM

Leveraging Semantic Signatures for Bug Search in Binary Programs

Jannik Pewny, Felix Schuster, Christian Rossow,
Lukas Bernhard, Thorsten Holz
Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany

Lehrstuhl für Systemsicherheit

ACSAC, December 12th, 2014

TEDEM - Pinpoint Vulnerabilities

Tree-Edit-Distance-based Equational Matching

- You show us a bug in a binary
- We re-find that bug in (another) binary

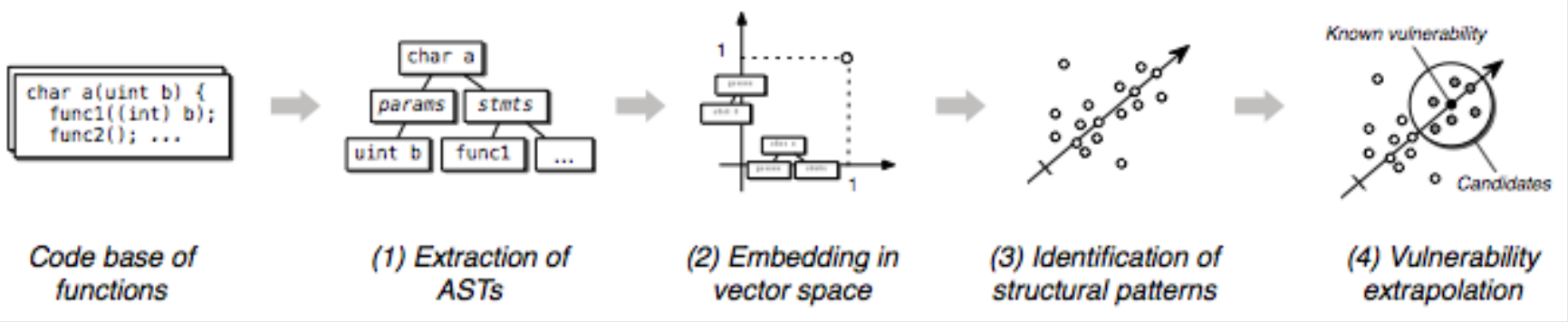
The basic assumption

- We know what a bug-instance looks like
- We find a piece of code, which looks very similar
- Likely, it contains the same bug

TEDEM - Pinpoint Vulnerabilities

Tree-Edit-Distance-based Equational Matching

- Inspired by: Generalized Vulnerability Extrapolation using Abstract Syntax Trees (ACSAC 2012)
- Fabian Yamaguchi, Markus Lottmann, Konrad Rieck
- Analyse source code, embed AST into vector space
- Pick a bug, use metric of that space to find similar code vectors



TEDEM - Pinpoint Vulnerabilities

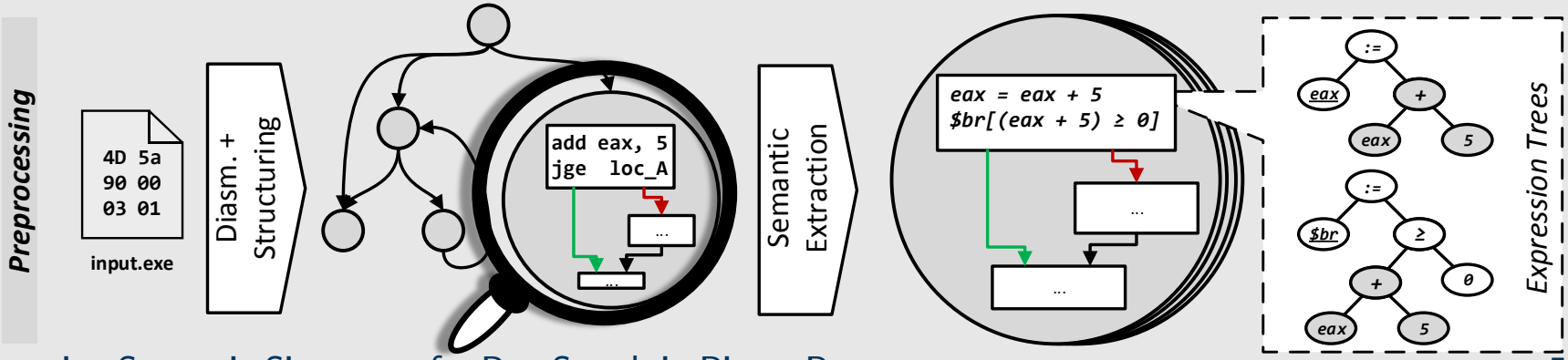
What we want

- Bug instances instead of bug classes
- Nearest neighbors instead of clustering
- Binary instead of source code
  - otherwise: cross-binary is hard
- Respect semantics/structure (rather than synthax)
- Sub-function granularity
- Focus on bug-relevant parts

TEDEM - Pinpoint Vulnerabilities

Preprocessing

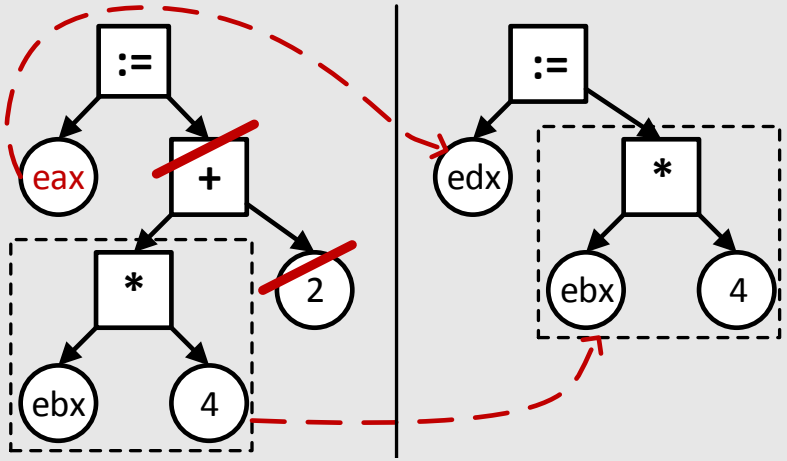
- Analyze the binary (Disassembly, Basic Blocks, CFG)
- *Real* Basic Blocks: Split at calls
- BB: Input-State $\mapsto$ Output-State
  - Aggregates and normalizes "Effect of BB on state"
  - Data-Structure: Tree
- Signature (what we search): CFG of BB-Input/Output-Equations
- Target (*in* what we search): CFG of BB-Input/Output-Equations



TEDEM - Pinpoint Vulnerabilities

Tree Edit Distance

- Basic Blocks are now essentially trees
- DISCLAIMER: No fully semantic comparison
  - But: Syntactic comparison, based on condensed, abstracted and normalized semantics
- Efficient XML Structural Similarity Detection using Sub-tree Commonalities (SBBD 2007)
- Tekli, Chbeir, Yetongnon

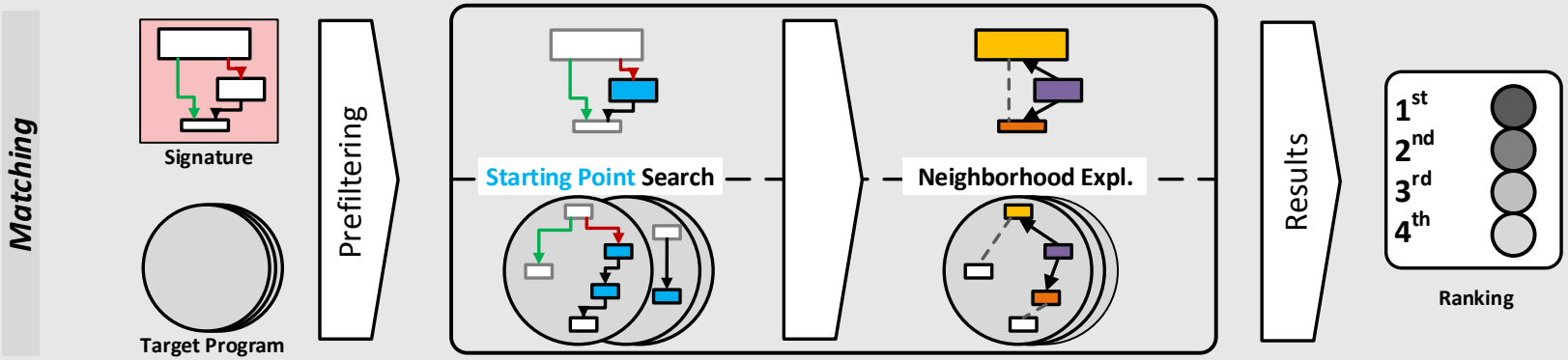


TEDEM - Pinpoint Vulnerabilities

Matching

- $\forall$ BB $\in$ Signature: Find BB-level starting points in target
  - Prefilter: Use heuristics; e.g. tree-size
  - Tree-Edit-Distance (with subtree edits)
- $\forall$ starting points:

  Explore Neighborhood, broaden match along next-best candidate in CFG, until full signature is matched
  - Greedy, but locally optimal
- Sort by overall distance $\Rightarrow$ Ranking



TEDEM - Pinpoint Vulnerabilities

Our niche: Fine-grained binary code comparison for bug-finding

- Usually: Search for bigger units (i.e. whole functions/libraries)
- Usually: Less precise metric, rely on "statistic effects"

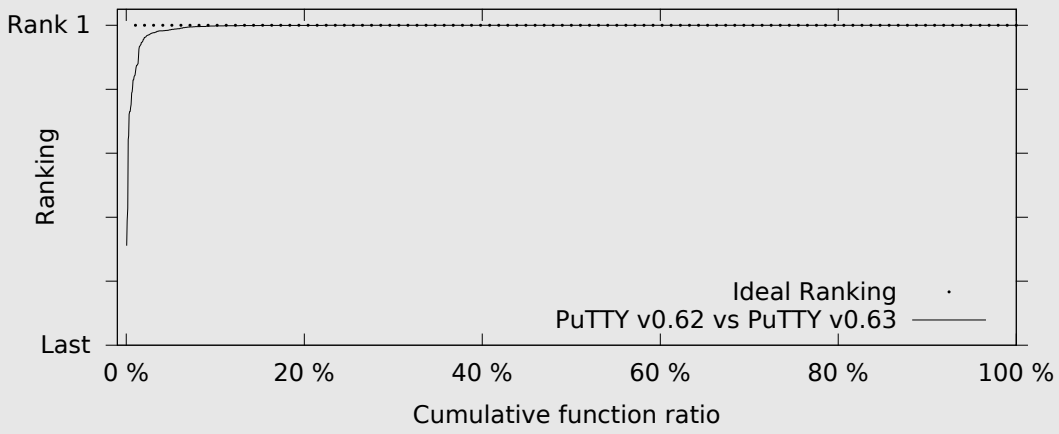
Results

- Pretty realiable for
  - Forks (ImageMagick/GraphixMagick, putty/tutty/puttycyg)
  - Shared Codebase (Pidgin/Adium;
    MS Word/Wordviewer/Compatibility Pack)
  - Multiple bug-occurrences in the same program
    - Results comparable to Yamaguchi's work *on source code*
- Sometimes even across OS-/compiler-borders
- Ironically, sometimes better than our manual ground-truth

TEDEM - Pinpoint Vulnerabilities

Experiments

- Our use-case $\neq$ function-matching
- No weighting
  - ... of bug-relevant/-irrelevant (Neighborhood Exploration)
  - ... of boilerplate-change vs. relevant change (Tree Edit Distance)
- No symbols, except to establish ground-truth
- On as-found-in-the-wild binary code; up to 400,000 BBs



TEDEM - Pinpoint Vulnerabilities

Runtime

- Preprocessing: One-time effort
  - Linear in |Program|
- Search for starting points
  - Linear in |Signature|
  - Linear in |Target| (Reduced to 5-10% with prefiltering)
  - Dominates runtime
- Tree Edit Distance
  - Quadratic in |Nodes|
- Neighborhood Exploration
  - Linear in |Signature|
  - Linear in #Neighbors of matched BBs

TEDEM - Pinpoint Vulnerabilities

Extrapolation

- Metric shows a degree of deviation
- Equal $\neq$ similar
- We can not only find identical bugs, but similar bugs

True positive?

- We find similar code. Is it really a bug?
- Would require much more complex analysis
- Example: Patched Code

TEDEM - Pinpoint Vulnerabilities

Patched Code

- Patched code is standard case of false positive
- Use two signatures:
  - one for the unpatched code
  - one for the patched code
- Higher similarity to patched signature

$\Rightarrow$ Likely also patched

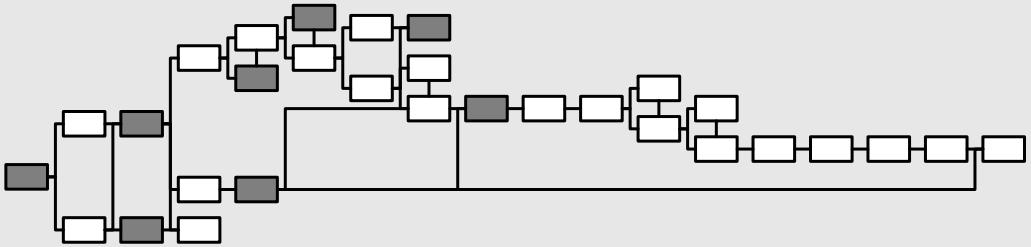
- Higher similarity to unpatched signature

$\Rightarrow$ Likely also unpatched

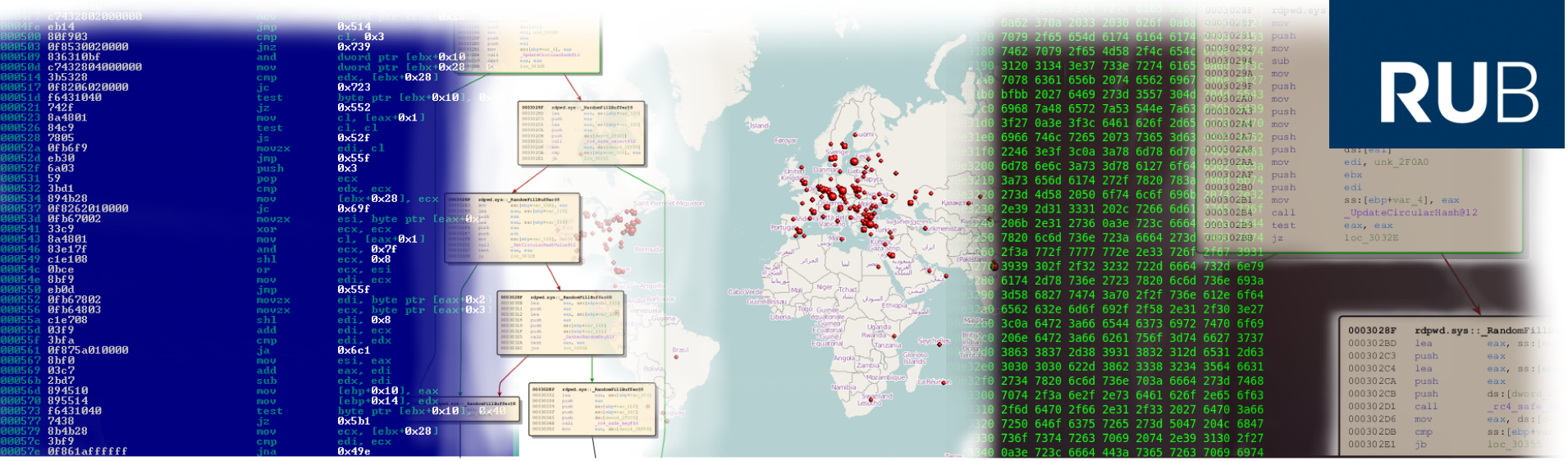
Limitations

Lessons learned

- Search for starting points dominates runtime
  - KD-Tree, VP-Tree, M-Tree did not work
- Neighborhood-relation needs to hold
  - Gaps or Recursive neighborhood exploration slow
- Based on similarity ⇒ Deviation affects results
  - Compiler, Optimization, Environment, ...



Good BB-matches in the two functions vulnerable to the Aurora-Bug



RUHR-UNIVERSITÄT BOCHUM

Many thanks for your attention!

Questions?

Jannik Pewny, Felix Schuster, Christian Rossow,
Lukas Bernhard, Thorsten Holz
<firstname>.<lastname>@rub.de

hgi Lehrstuhl für Systemsicherheit