HI-CFG:

Construction by Dynamic Binary Analysis, and Application to Attack Polymorphism

Dan Caselden, Alex Bazhanyuk,
Mathias Payer, Stephen McCamant,
Dawn Song, UC Berkeley

Recovering Information

Knowledge of information (data) flow and control flow of an application crucial for analysis

- Current tools focus on just one type of flow

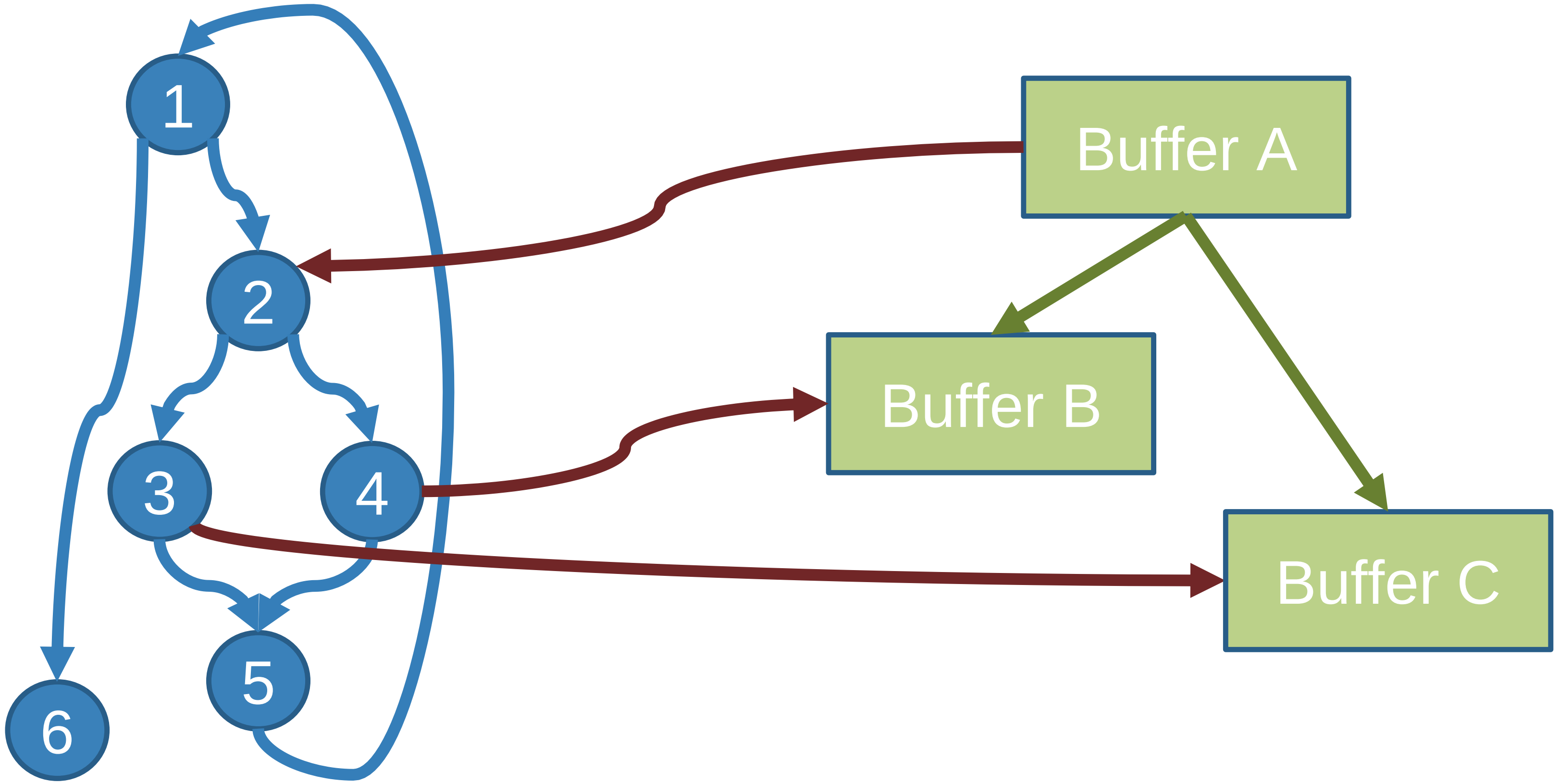
Combine information flow and control flow into high-level data structure

- Hybrid, Information- and Control-Flow-Graph (HI-CFG) using binary analysis

HI-CFG Overview

CFG view

Data flow view



Outline

Motivation

Attack Polymorphism

Dynamic HI-CFG Construction

Evaluation

Conclusion

HI-CFG: Attack Polymorphism

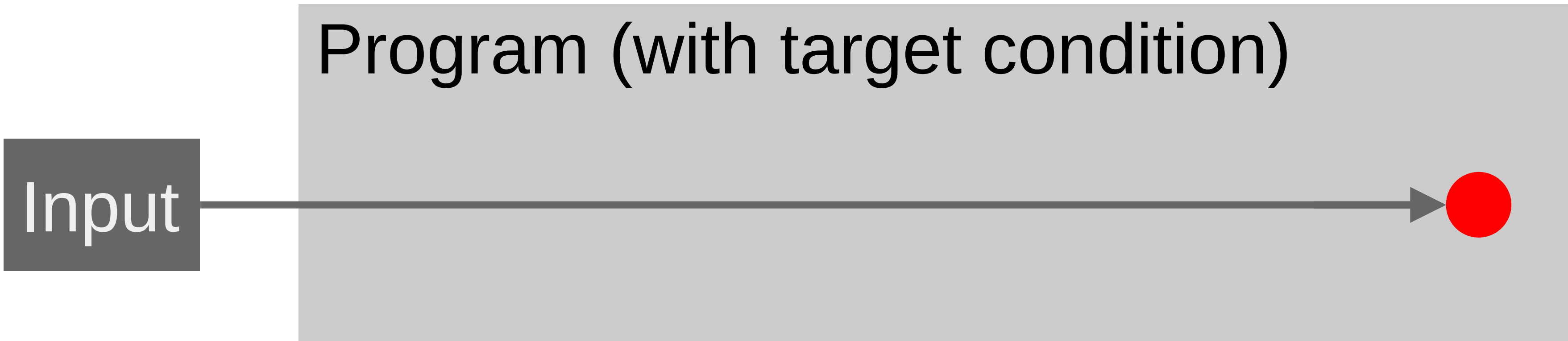
Step one: phase partitioning

- Divide a computation into steps that transform data from an original input to an internal format
- Based on HI-CFG buffers, information-flow and producer/consumer edges

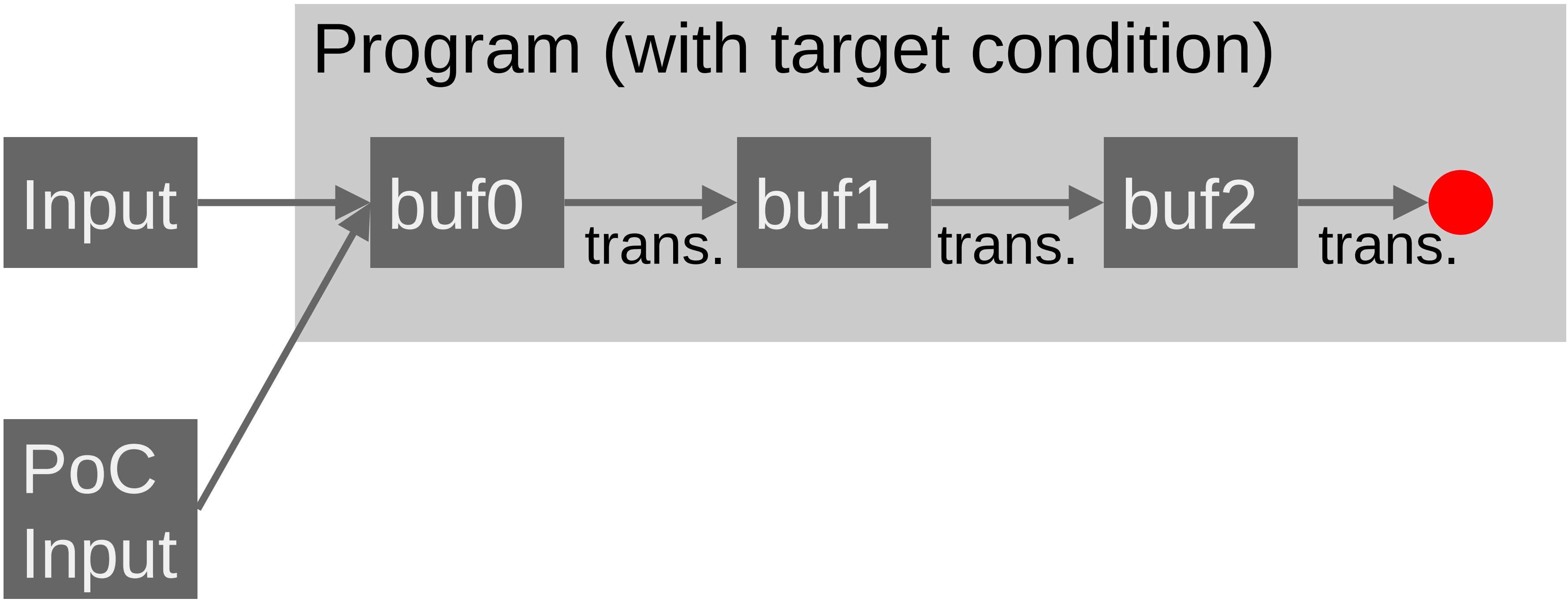
Step two: phase aware input generation

- Aim is to produce an input that triggers a vulnerability deep within a program
- Use phase structure to divide and conquer
- Symbolic execution with search pruning

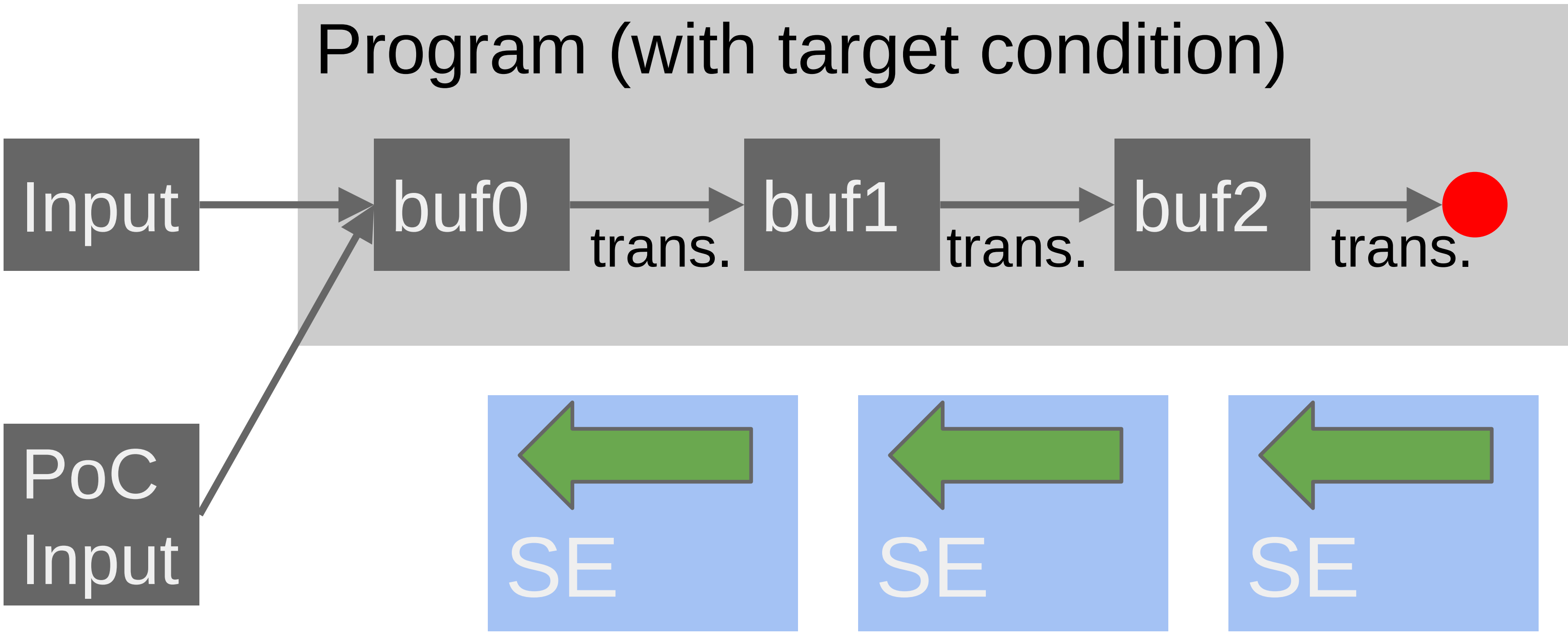
HI-CFG: Attack Polymorphism



HI-CFG: Attack Polymorphism



HI-CFG: Attack Polymorphism



Outline

Motivation

Attack Polymorphism

Dynamic HI-CFG Construction

Evaluation

Conclusion

HI-CFG: trace-based construction 1/3

Trace enables us to recover both control-flow and information-flow of an application using some concrete input

1. Start with specific input data
2. Collect an instruction level trace (TEMU)
3. Process the traces to create a HI-CFG

HI-CFG: trace-based construction 2/3

Work through the execution trace and group "***related***" memory accesses

- Categorize buffers hierarchically
- Conservative and taint-based information flow

Grouping heuristics

- Instructions use same base pointer
- Temporally and spatially correlated memory accesses

HI-CFG: trace-based construction 3/3

Apply graph partitioning algorithms to divide the HI-CFG at "***natural***" boundaries to separate code and data structures

- Extract functionality into separate modules for reuse or transformation

No source info needed, except addresses of `malloc/calloc/free`

Outline

Motivation

Attack Polymorphism

Dynamic HI-CFG Construction

Evaluation

- Scalable Symbolic Execution
- Poppler Case Study

Conclusion

Scalable SE is key

Vulnerability detection

- Both in malware and legit applications

Model extraction

- Automatically learn security-relevant models

Binary code reuse

- Identify interface and extract components

Evaluation setup

Simple transformation

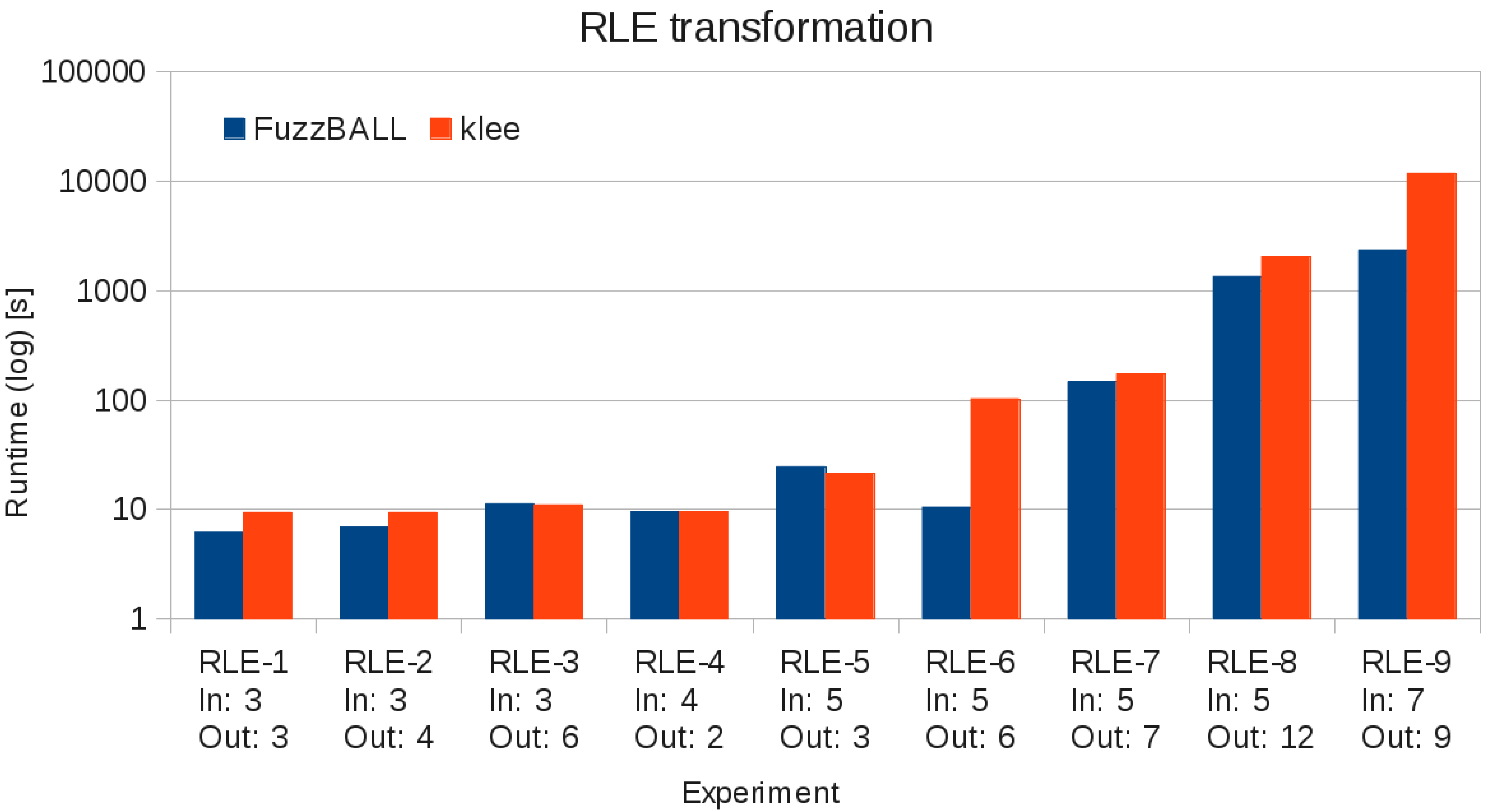
- RLE decoding
- Output as target, SE produces input

Configurations

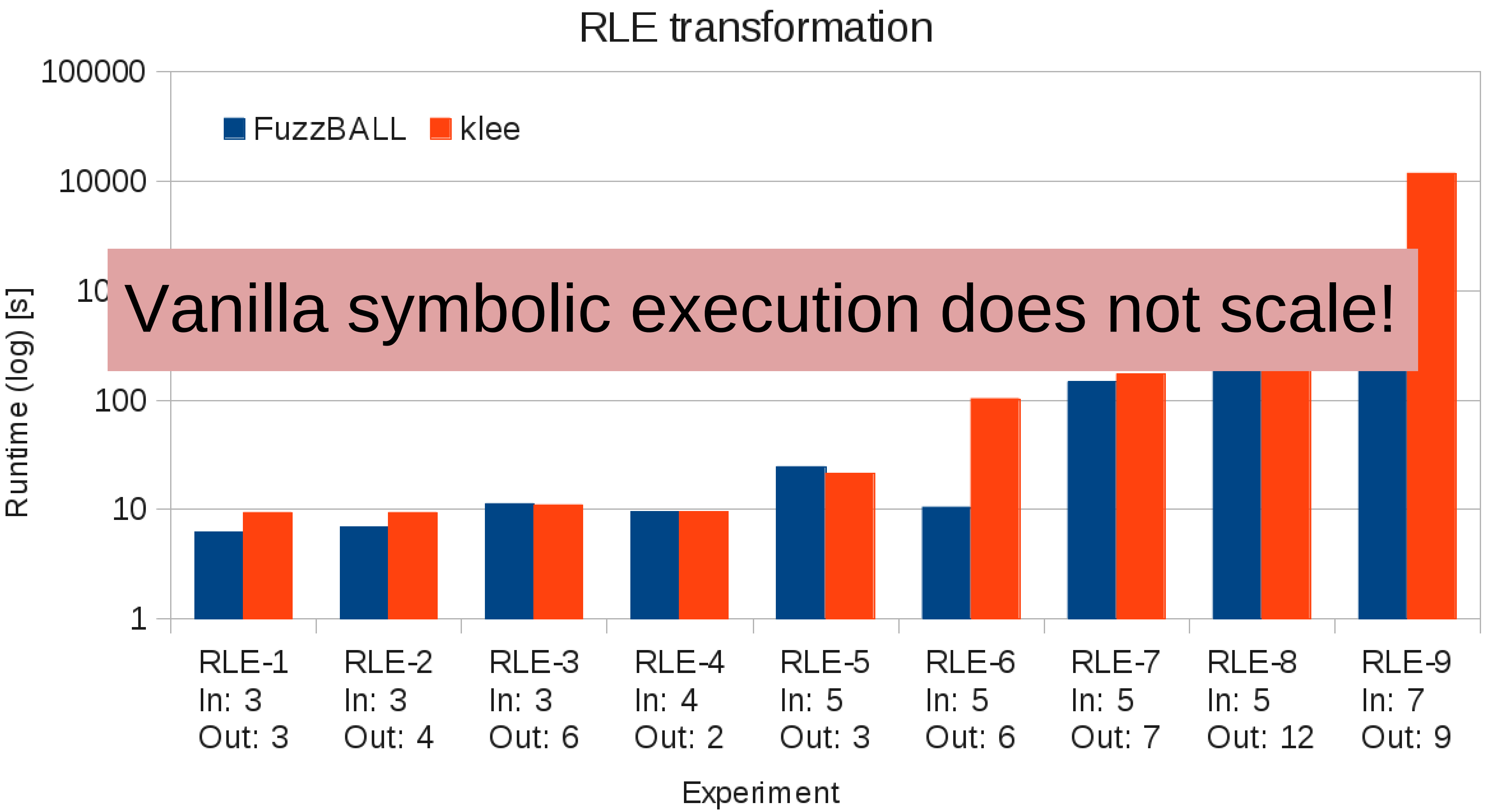
- KLEE
- FuzzBALL

Detailed results from TR Berkeley/EECS-2013-125

Limitations of SE



Limitations of SE



Vanilla symbolic execution does not scale!

Transformation-aware SE

Computations rely on input transformations

Focus on transformations to reduce complexity

- Surjectivity guarantees existing pre-image
- Sequentiality ensures output is never revoked
- Streaming bounds the transformation state

Covered transformations include decryption, decompression, escape sequences, image or sound decoding

Feedback-guided optimization (FGO)

Search pruning

- if target "***unreachable***"

Search prioritization

- look for short inputs that maximize size of output

Symbolic array accesses

- treat choice of index like a branch (baseline)
- combine all possible values into formula

Evaluation setup

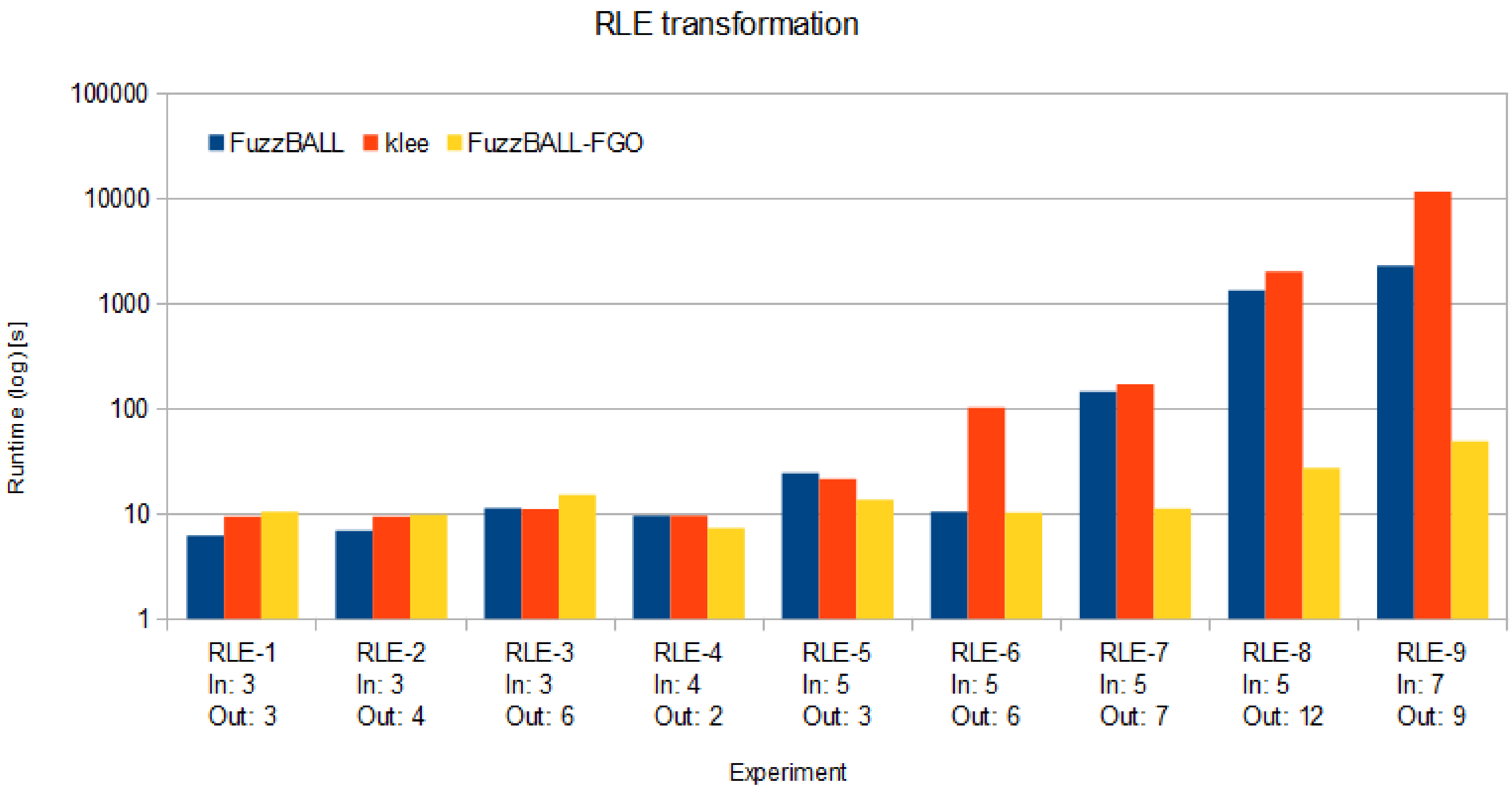
Simple transformation

- RLE decoding
- Output as target, SE produces input

Configurations

- KLEE
- FuzzBALL
- FuzzBALL-FGO

FGO: 1 order of magnitude



Transformation-aware SE

Divide-and-conquer strategy for SE

- HI-CFG captures transformations
- Split SE on transformation boundaries

Evaluation setup

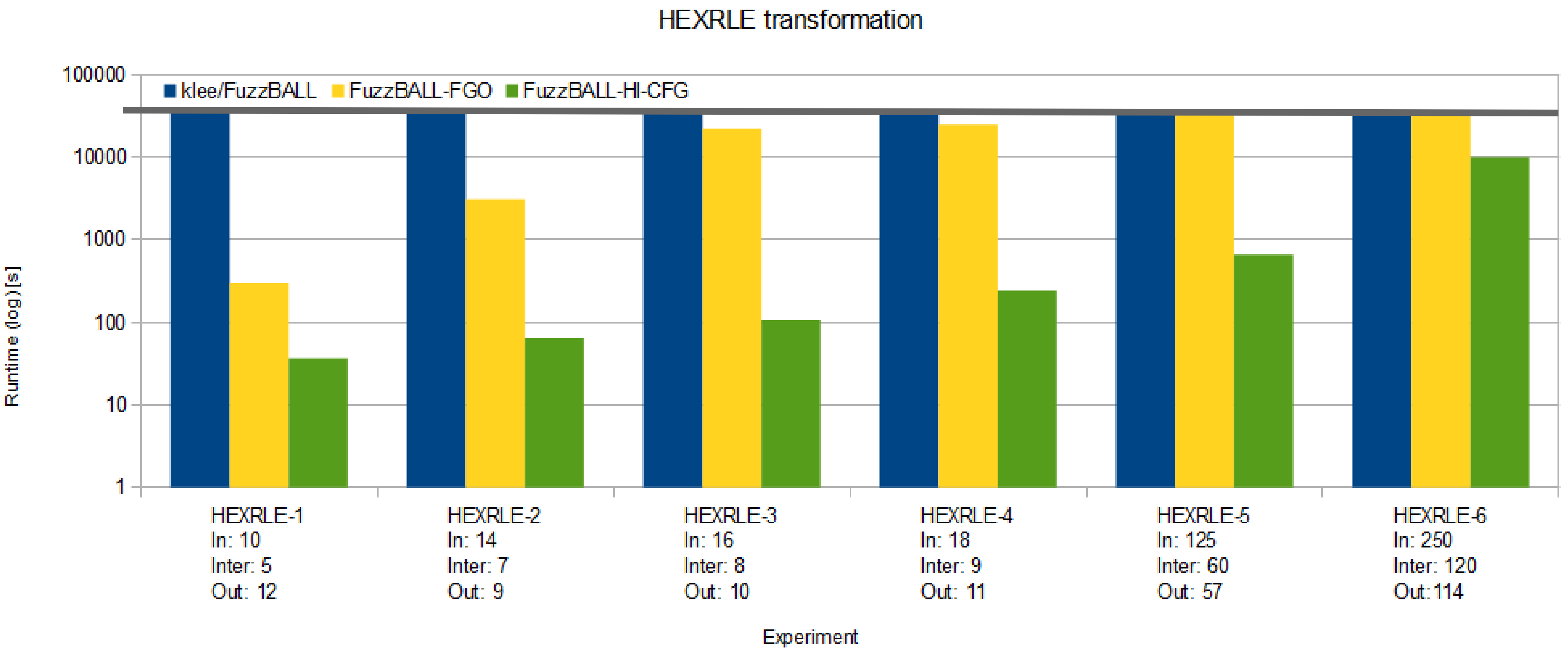
Two transformations

- HEX decoding
- RLE decoding

Different configurations:

- KLEE/FuzzBALL
- FuzzBALL-FGO
- FuzzBALL-HI-CFG (includes FGO)

Transformation-aware SE: another 1 order of magnitude



Poppler Case Study

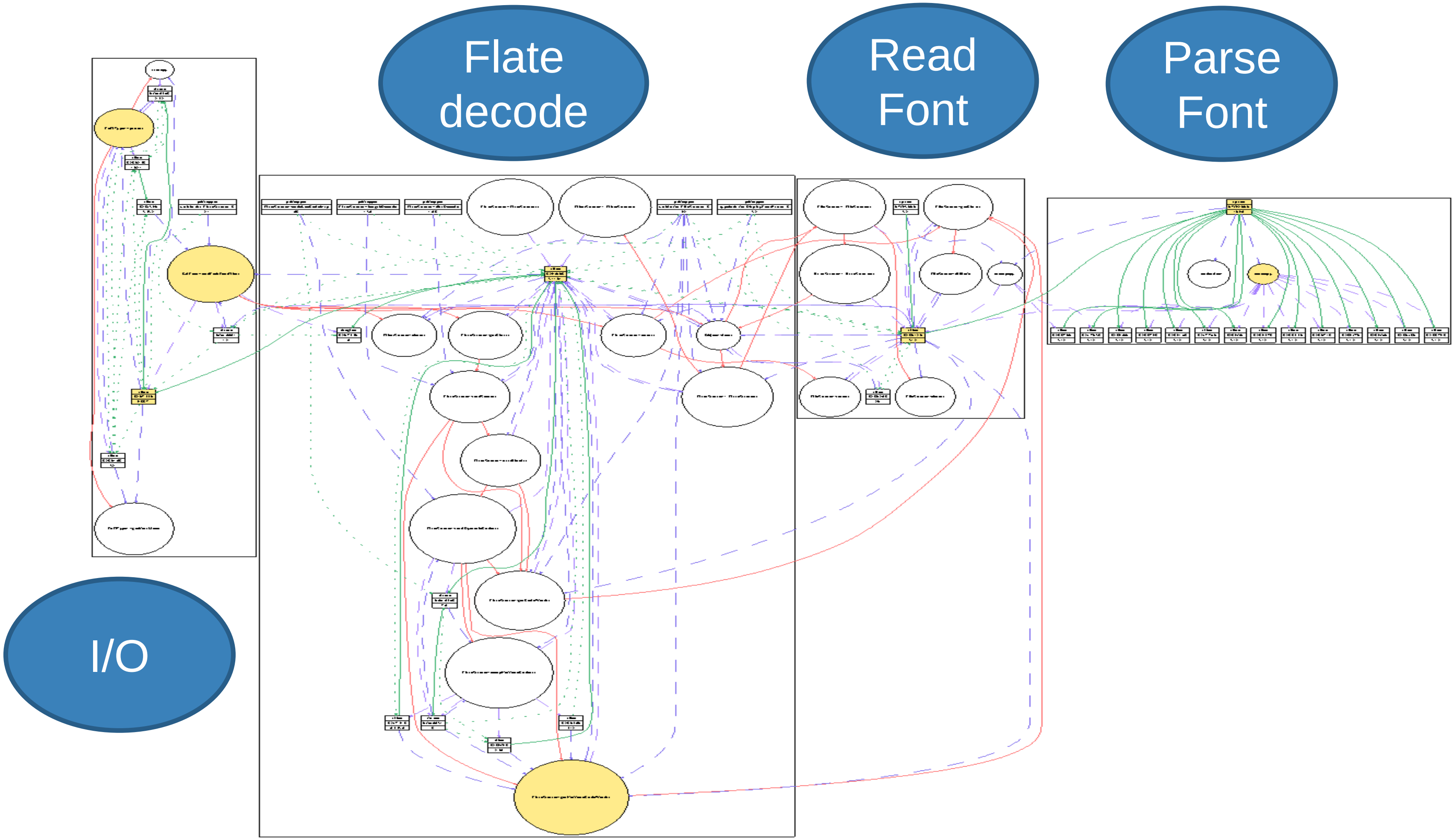
Poppler PDF viewer

- Type 1 font parsing vulnerability CVE-2010-3704

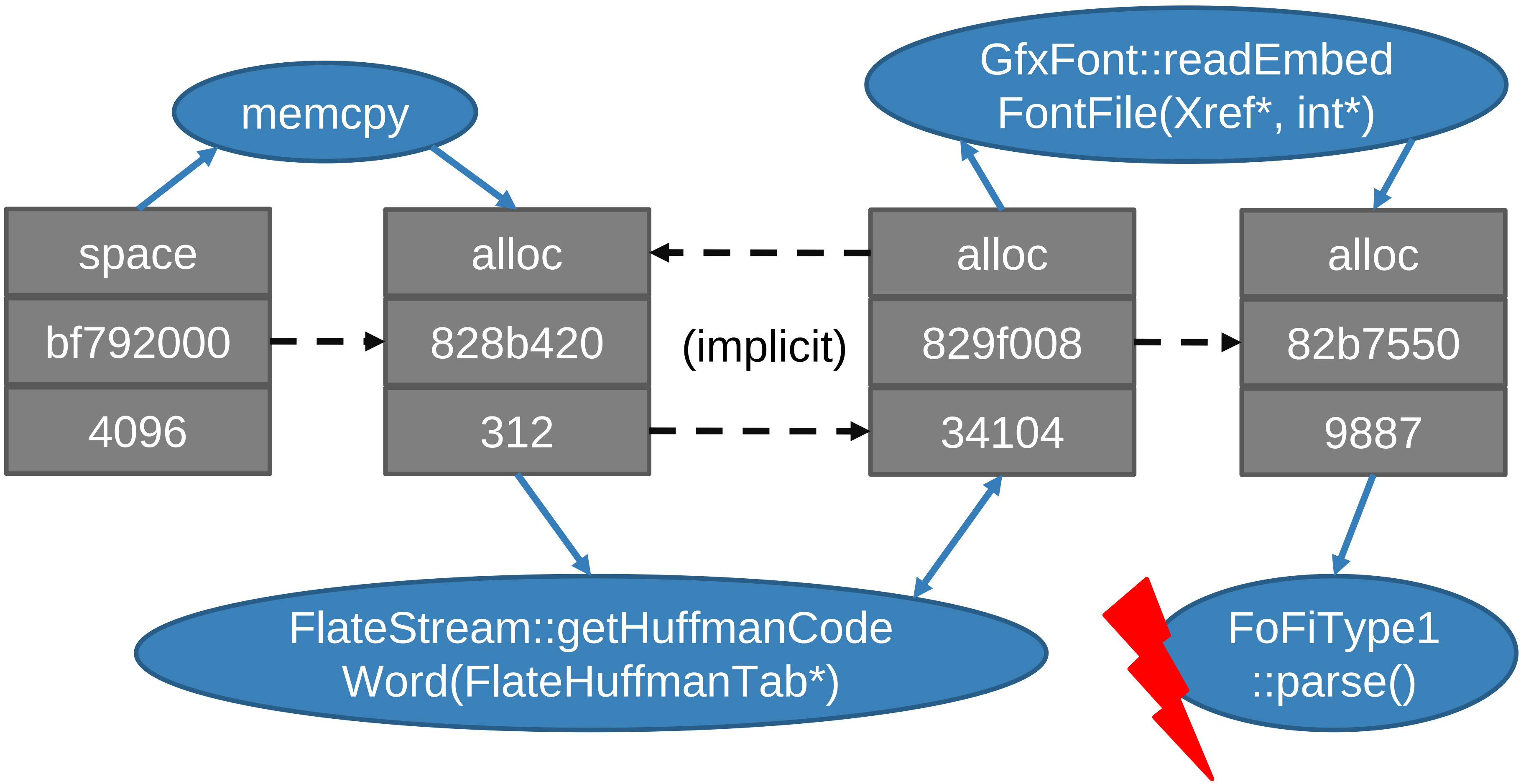
HI-CFG construction using benign document that loads a font

- PDF generated by pdftex using a small tex file

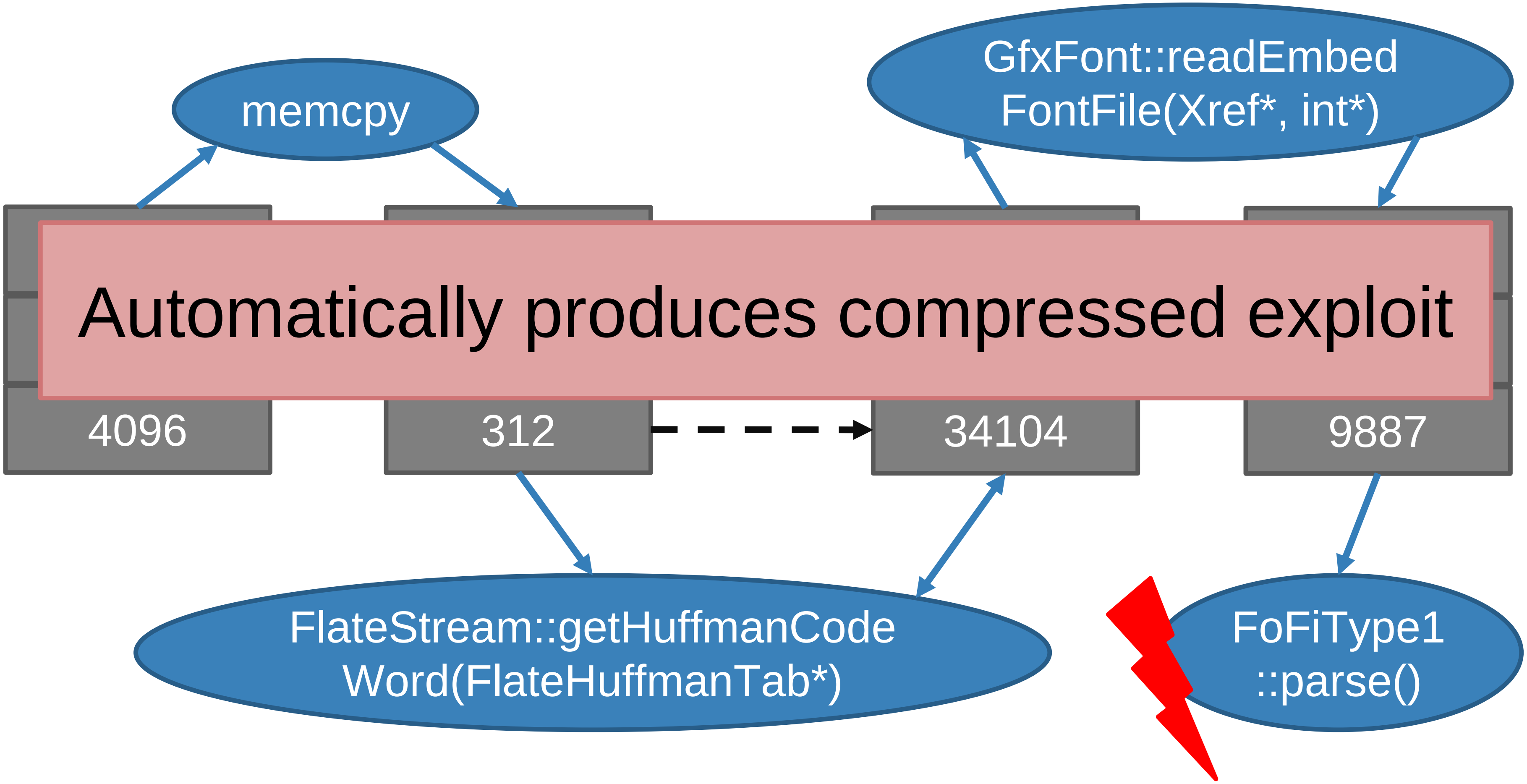
Poppler Phases



Poppler Buffers



Poppler Buffers



Outline

Motivation

Attack Polymorphism

Dynamic HI-CFG Construction

Evaluation

Related Work

Conclusion

Related Work

HOWARD (Slowinska et al., NDSS’11, ATC12):
Type and data structure inference from binaries

- HI-CFG looks at code & relationships between code and data (not just data structures)

AEG (Avgerinos et al., NDSS’11) and
MAYHEM (Cha et al., Oakland’12):
SE-based attack input generation

- HI-CFG enables focus on iterative and scalable SE (not focus on coverage)

Outline

Motivation

Attack Polymorphism

Dynamic HI-CFG Construction

Evaluation

Related Work

Conclusion

Conclusion

Presented HI-CFG as new data-structure

- Construction from binary execution traces

HI-CFG enables

- Deep program analysis
- Recover components from binaries
- Guide SE along probable paths

FuzzBALL symbolic execution engine:

- http://github.com/bitblaze-fuzzball/fuzzball