Automatic Software Error Finding: Approaches and Tradeoffs

Vaggelis Atlidakis

PEOPLE AFFECTED (AT LEAST)

3,683,212,665

SOFTWARE FAIL WATCH

5TH EDITION

2017

LO$$E$

1,715,430,778,504

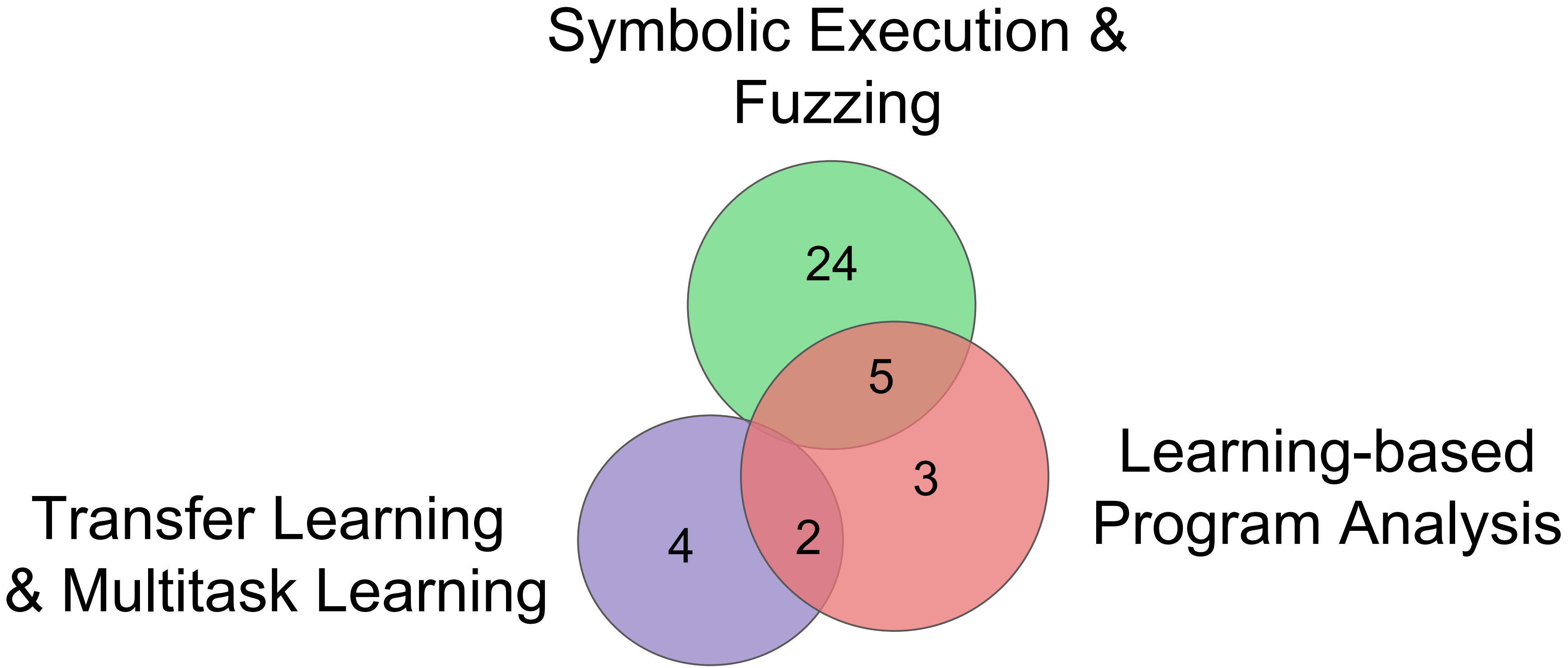
TRICENTIS

Software errors: Find them before they find you!

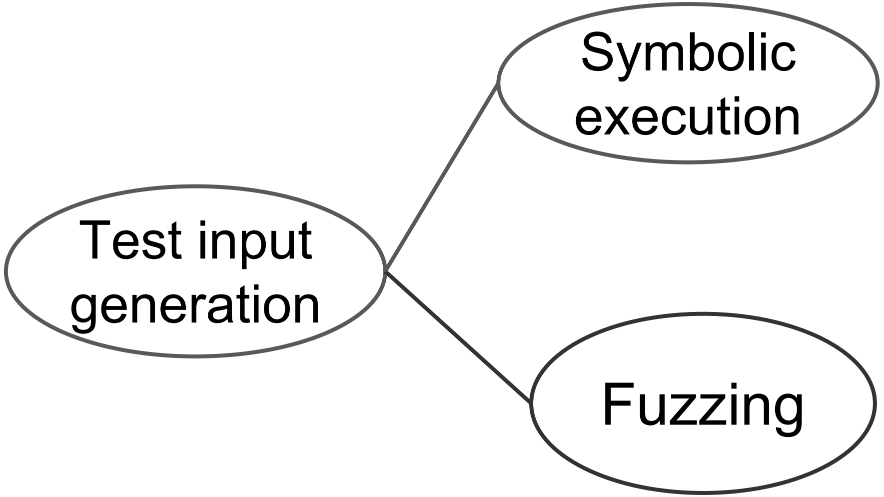
Reviewed approaches

- Test input generation
- Statistical error detection

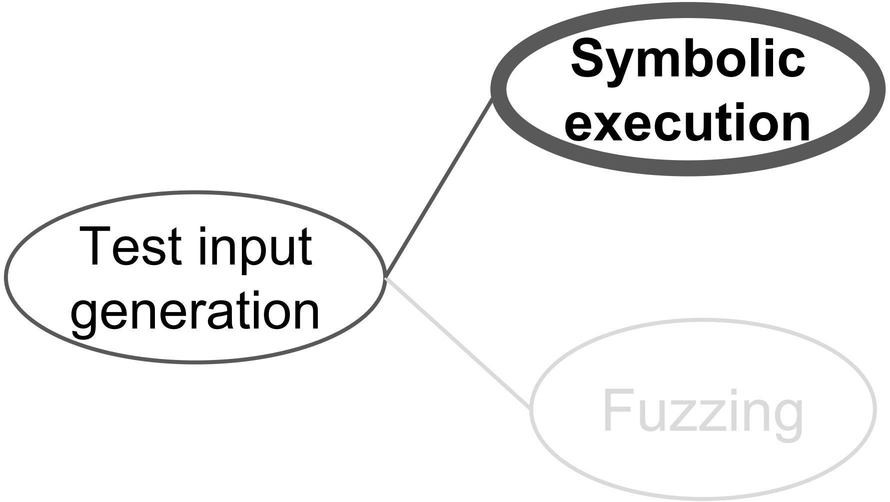
Papers from three areas



Symbolic execution & fuzzing



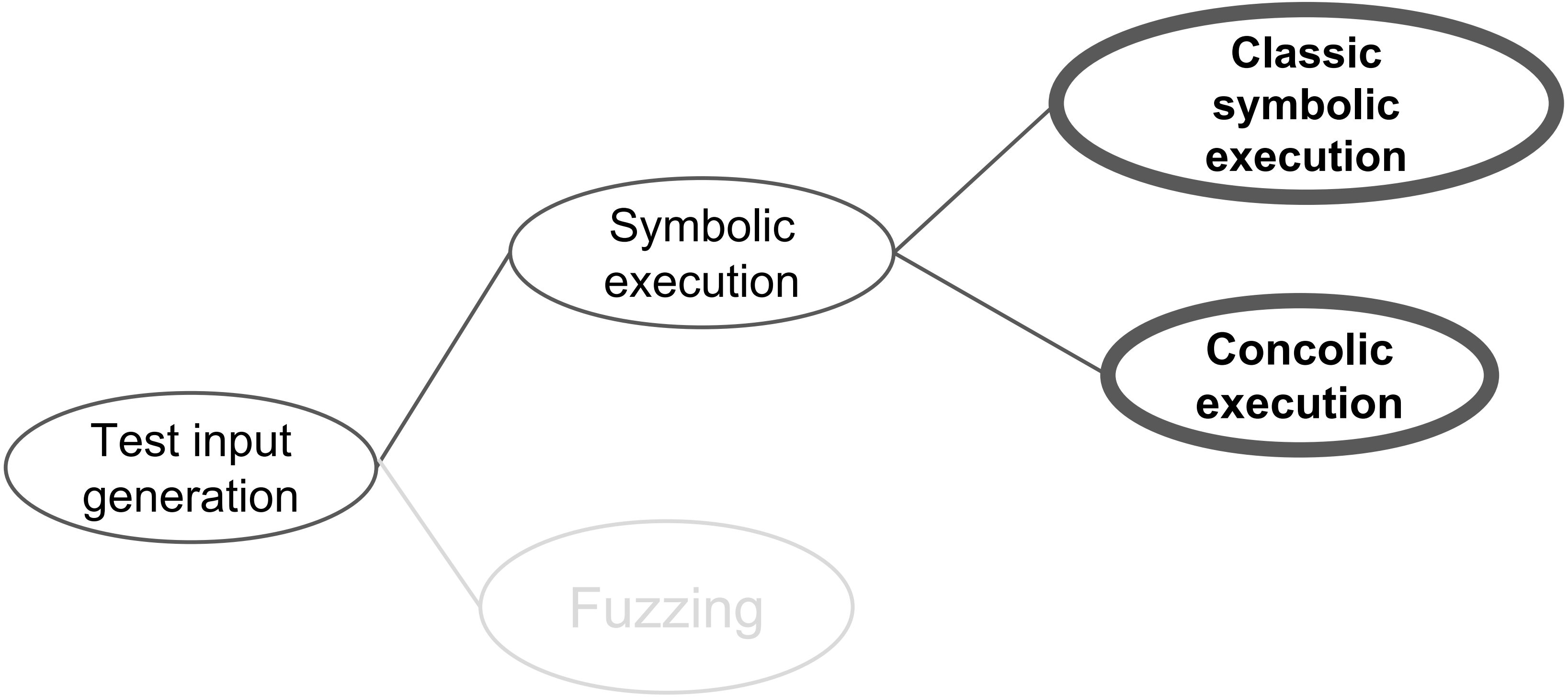
Symbolic execution 101



Introduced in EFFIGY[1](1976)

- Execute on symbolic inputs
- Summarize classes of inputs

Classic symbolic vs concolic execution



Classic symbolic vs concolic execution

Classic symbolic execution

- Maintain symbolic state
- Fork symbolic execution on branches
- Use solver
    - Branch feasibility

➢ Implemented in EXE [2](2006)

Concolic execution

- Maintain **conc**rete & symb**olic** state
- Run concrete execution on taken branches
- Use solver
    - Cover not-taken branches

➢ Implemented in DART [12](2005)

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

  if (z == y + 1)
        abort();
  else
        exit(0);
}
```

Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		Random concrete values: x=1, y=1
z=a*a*a		z=1
	a*a*a != b + 1	

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

  if (z == y + 1)
        abort();
  else
        exit(0);
}
```

Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		Random concrete values: x=1, y=1
z=a*a*a		z=1

Path constraint: (a*a*a != b + 1)

Cannot solve

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

    if (z == y + 1)
          abort();
    else
          exit(0);
}
```

Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		Random concrete values: x=1, y=1
z=a*a*a		z=1

Path constraint: (a*a*a != b + 1)

Cannot solve

Simplify: a = 1 ↔ 1 != b + 1

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

  if (z == y + 1)
        abort();
  else
        exit(0);
}
```

Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		**New concrete values: x=1, y=0**
z=a*a*a		z=1

Path constraint: (a*a*a != b + 1)

Cannot solve

Simplify: a = 1 ↔ 1 != b + 1

Negate & solve: b = 0

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

  if (z == y + 1)
          abort();
  else
          exit(0);
}
```

Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		New concrete variables: x=1, y=0
z=a*a*a		z=1
	a*a*a == b + 1	

Example

```
void test_me(int x, int y){
    //naughty programmer
    z = syscall(x)

   if (z == y + 1)
         abort();
   else
         exit(0);
}
```

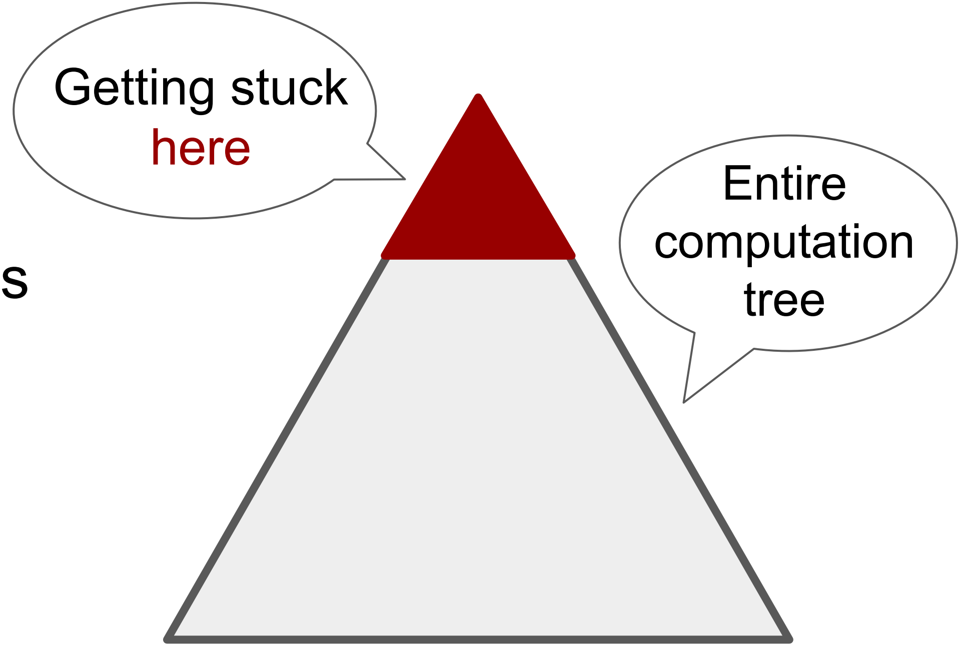
Symbolic state	Path constraint	Concrete state
Create symbolic variables: x=a, y=b		New concrete variables: x=1, y=0
z=a*a*a		z=1
	a*a*a == b + 1	

Classic symbolic execution

General limitations

- Handling complex constraints
- Environment problem

➢ Path explosion

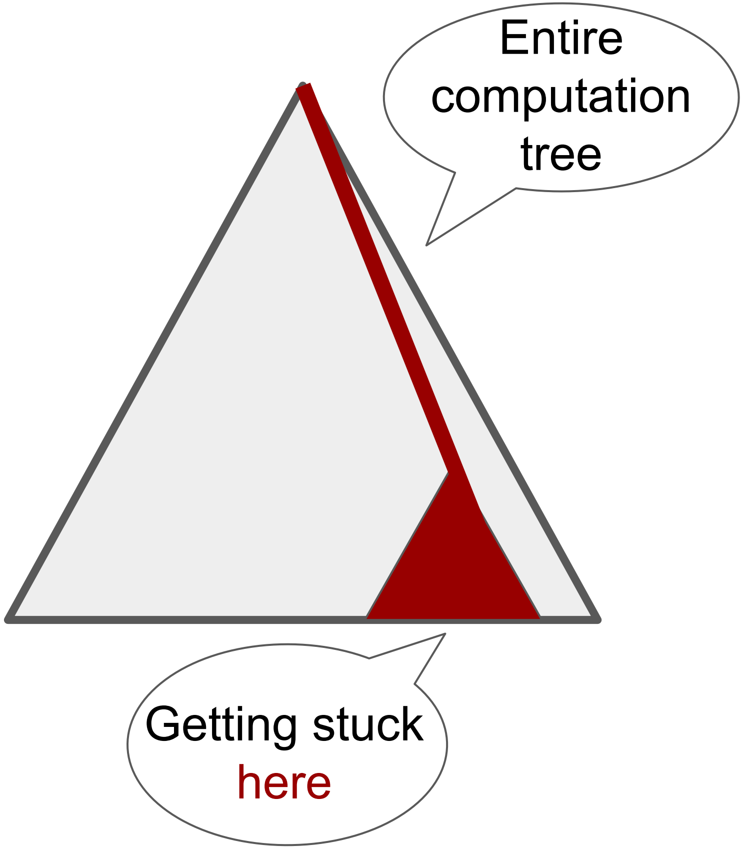


Concolic execution

What do we gain?

- Executions run to completion

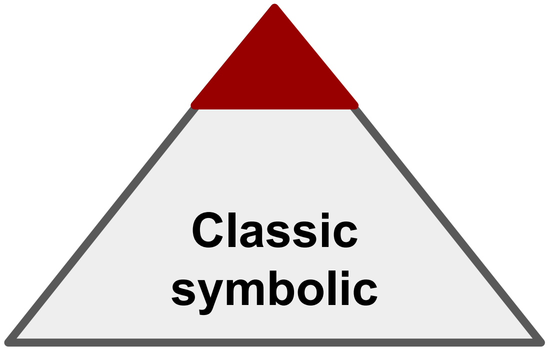
➢ Path explosion still a problem



Entire computation tree

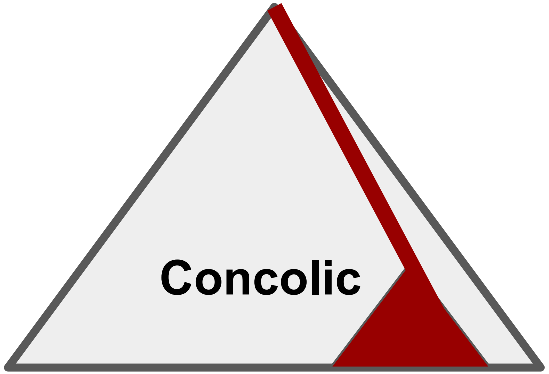
Getting stuck here

Comparative view



Coverage	Number of COREUTILS tools	Avg. #ELOC
100%	16	3307
90-100%	38	3958
80-90%	22	5013

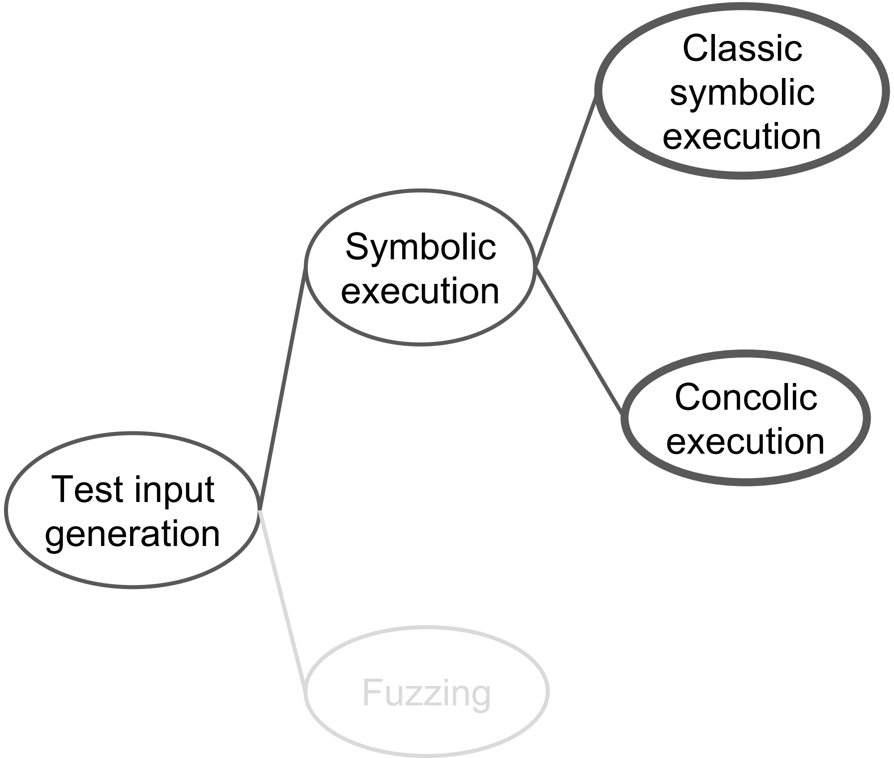
COREUTILS tools statement coverage KLEE [3]



App	Mean number of instructions	#Test cases
Media	54M	2,266
Office	923M	3,008

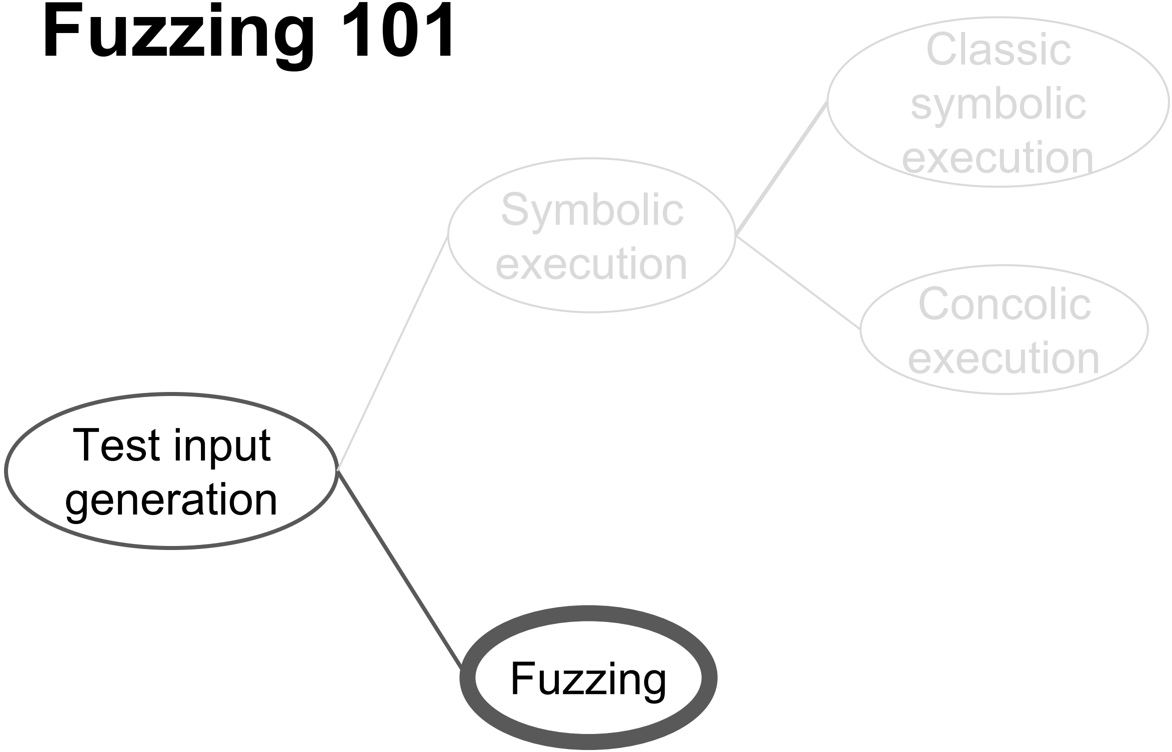
Statistics from SAGE[19]

Classic symbolic vs concolic execution

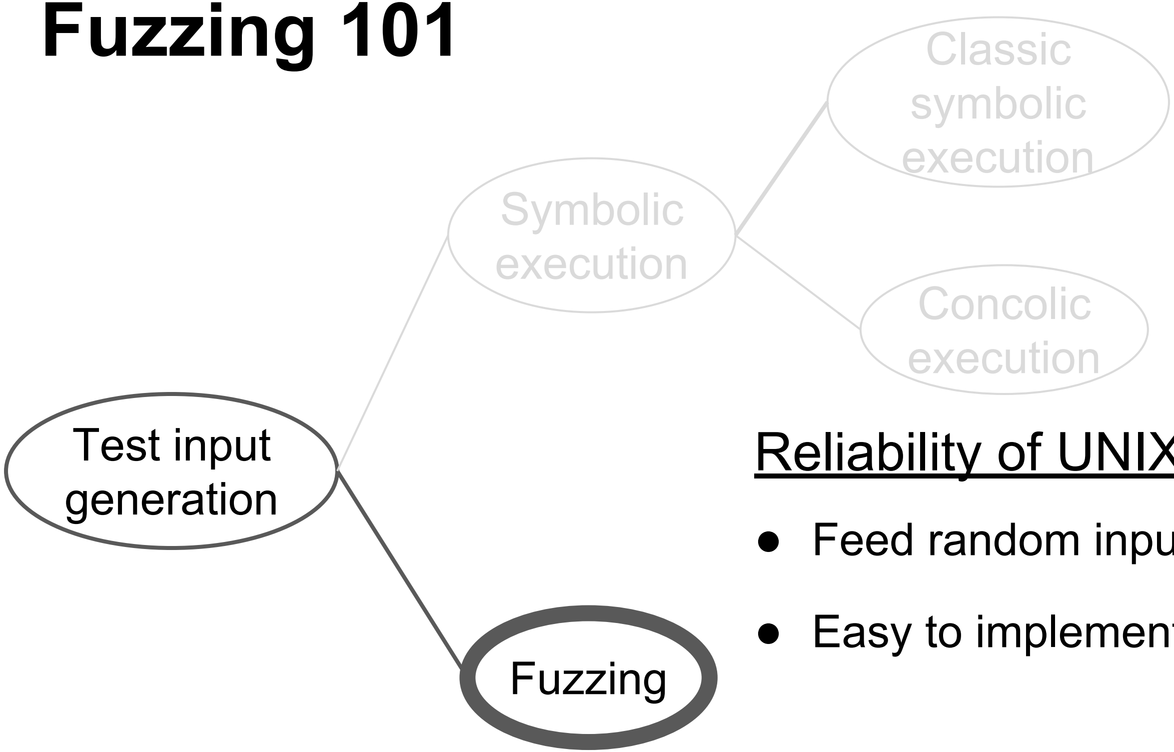


System	Type	What's new?
EXE[2]	Symbolic	Pioneer symbolic execution engine
KLEE[3]	Symbolic	Models environment
UC-KLEE[9]	Symbolic	Checks individual functions
CLOUD9[7]	Symbolic	Parallelization of symbolic execution
DART[12]	Concolic	Pioneer concolic execution engine
CUTE[13]	Concolic	Adds symbolic with pointers
PEX[5]	Concolic	Concolic execution in .NET
SAGE[19]	Concolic	Generational search on deep paths
CREST[15]	Hybrid	Concolic exec. & random testing
VERISOFT[8]	Hybrid	Concolic exec. & state merging
S2E[6]	Hybrid	Symbolic exec. w/ virtualization

Fuzzing 101


Test input generation
Symbolic execution
Classic symbolic execution
Concolic execution
Fuzzing

Fuzzing 101



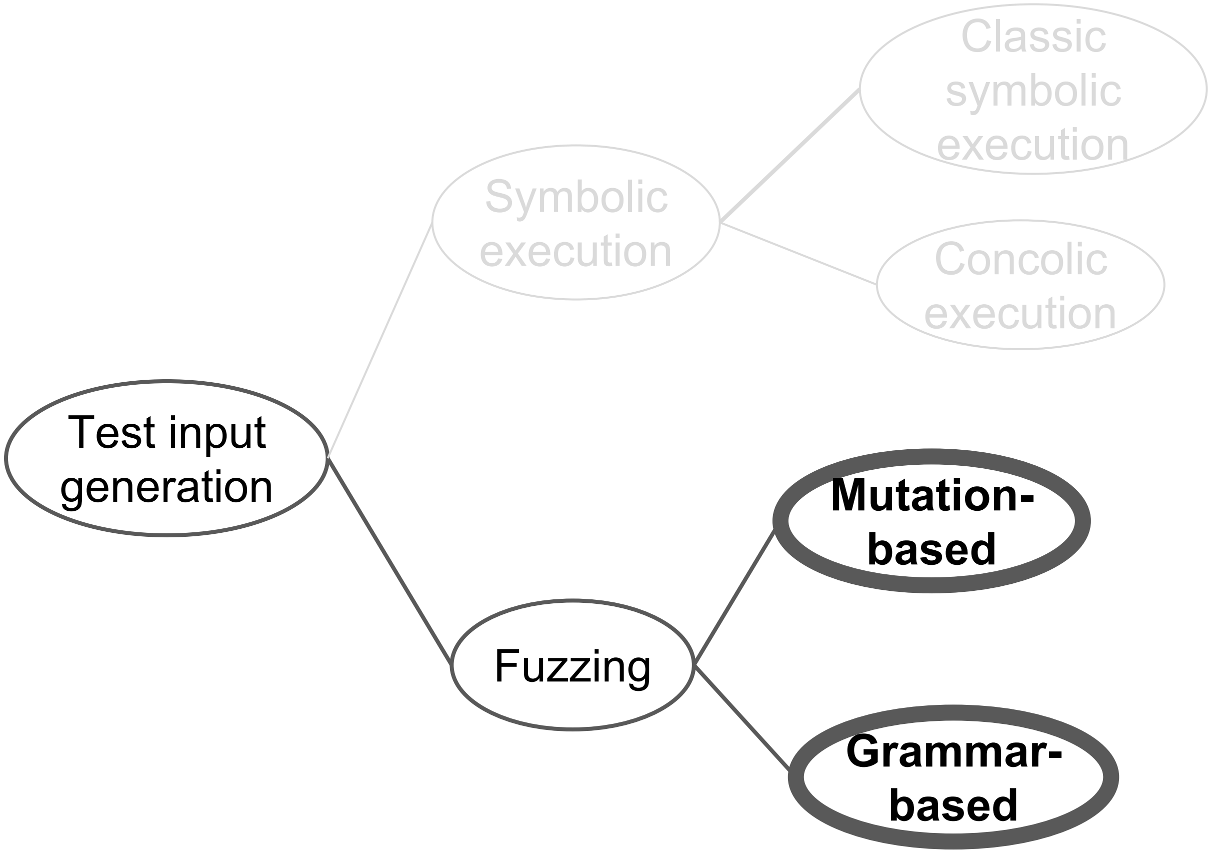
Reliability of UNIX utilities[18](1990)

- Feed random inputs and monitor for errors
- Easy to implement:

```
$> while true; \
  > do head -n 10 /dev/urandom | a.out; \
  > done
```

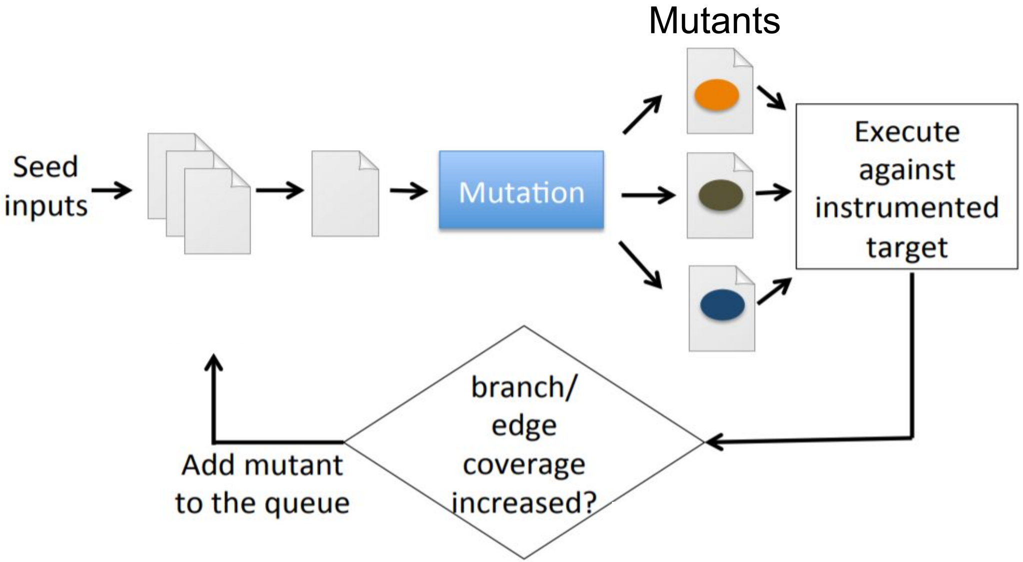
- Inputs that trigger incorrect behaviour are small fraction

Fuzzing: mutation-based vs grammar-based



Mutation-based fuzzing

- ➢ American Fuzzy Lop (AFL)[23]



Key points

- Coverage-guided search
- No assumptions for particular input format
- Hard branches (e.g., magic numbers)

Grammar-based fuzzing

➢ SPIKE grammar-based fuzzer [24]

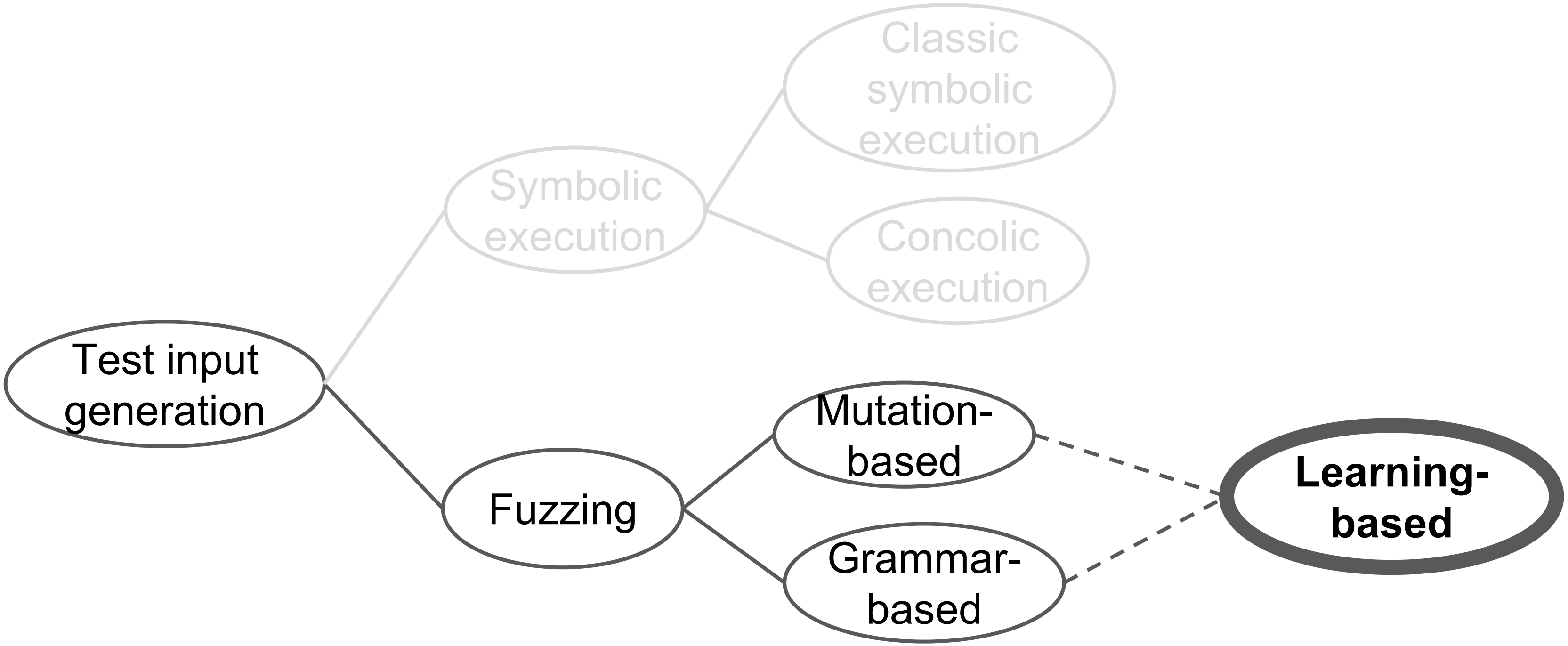
```
// Magic number -- don’t fuzz
static("89504E470D0A1A0A");
…
// Fuzz next bytes
block_start("Header");
fuzzable_byte(1);  // Width
...
block_end("Header”);
...
```

Key points

- Use grammar to describe input formats
- Good for structured input formats

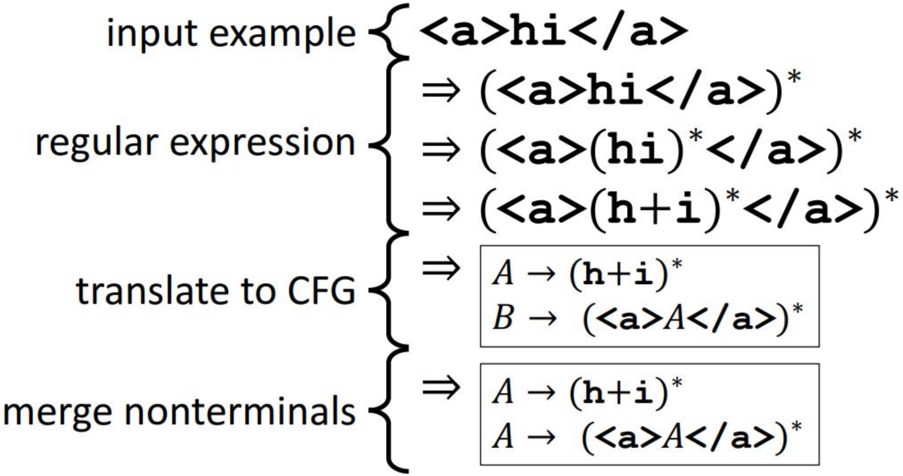
➢ Writing grammar is labour-intensive, manual process

Learning-based fuzzing



Learning-based fuzzing

➢ GLADE: Synthesizing program input grammars [25](2017)



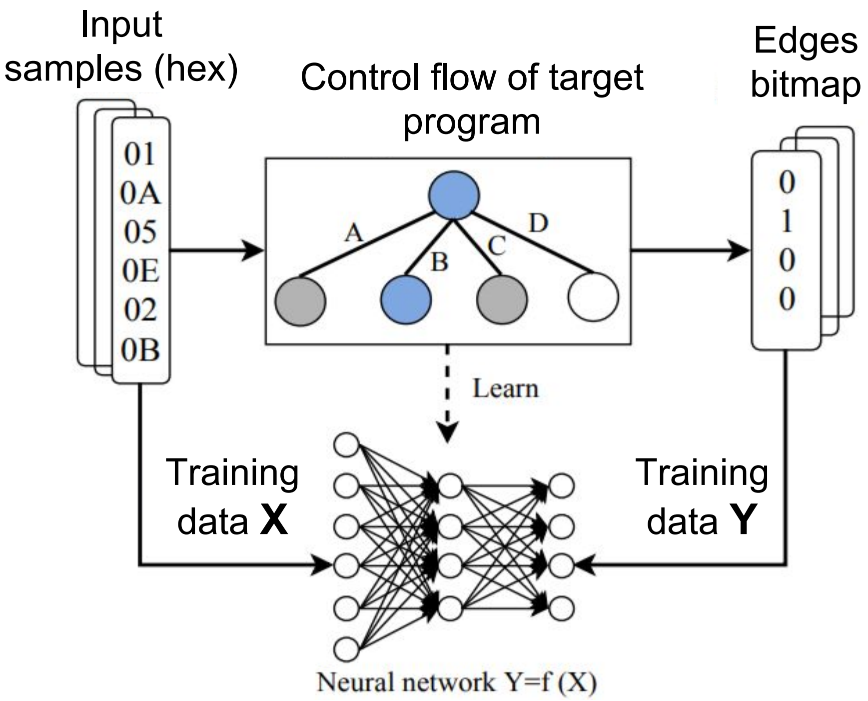
Key points

- Start with an input sample
- Construct increasingly general regular expressions
- Translate to Context Free Grammar

➢ Learning is slow

Learning-based fuzzing

➢ NEUZZ: Fuzzing with Neural Program Learning [29](2018)



Key points

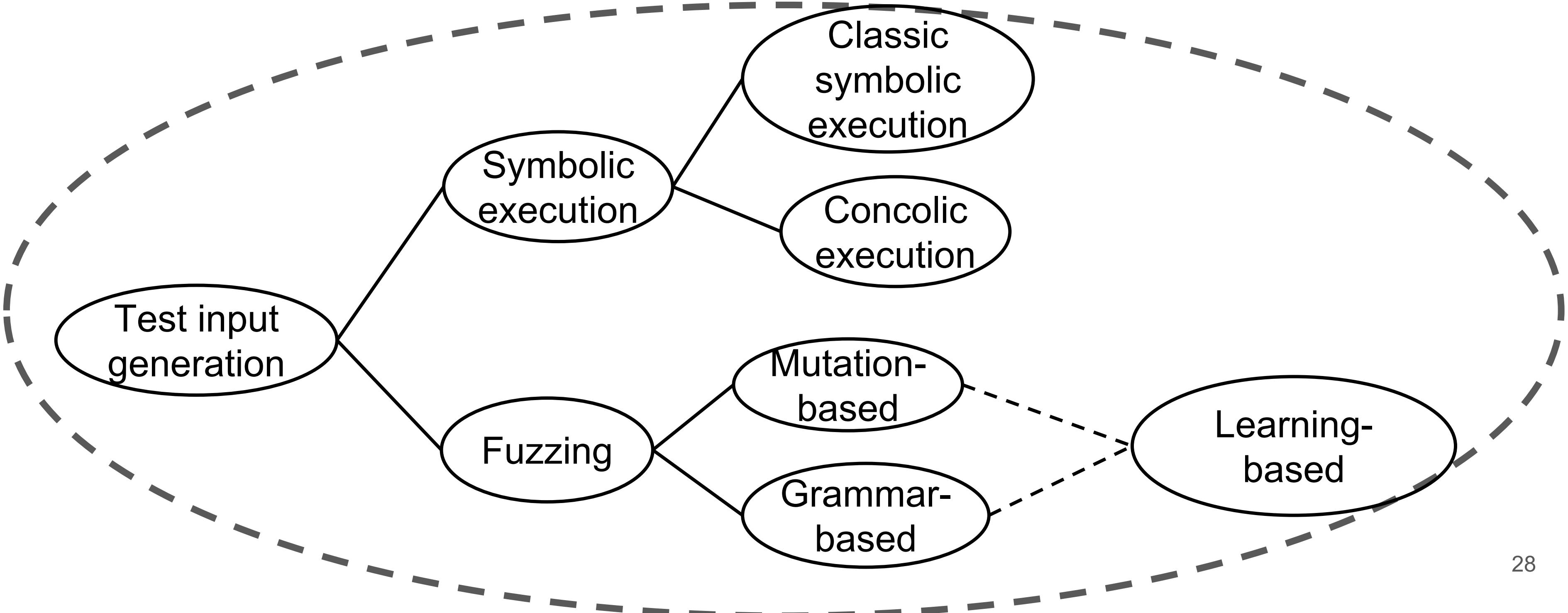
- Feed input samples and monitor taken/non-taken branches
- Use training data X,Y learn model for branching behavior
- Use model to perform gradient-guided mutations

➢ Unclear generalization to “never-taken” paths

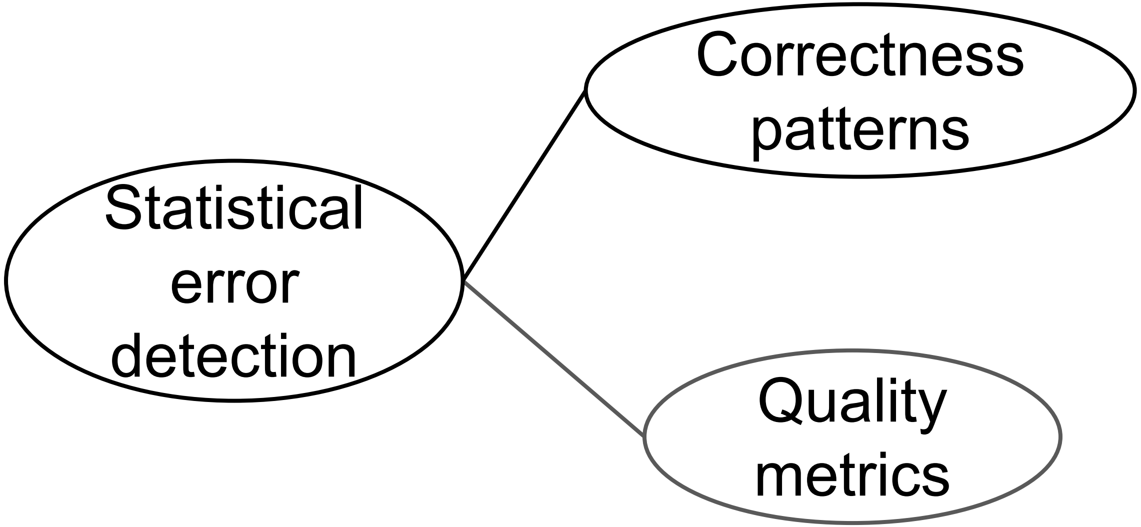
Learning-based fuzzing

Characteristics	**GLADE [25]**	**NEUZZ [29]**	**SKYFIRE [27]**	**RL Fuzzing [28]**	**Learn & Fuzz [26]**
Learns to model	Valid input format	Taken/non-taken branches	Valid input format	High reward mutation policy	Valid input format
Mutations	Use grammar	Use model's gradients	Use grammar and AFL	Use learnt policy	Use model's predictions
Strength	Fully blackbox	Gradient-guided mutations	Semantic validity of test cases	End-to-end RL formulation	Location-specific mutation probabilities
Weakness	Learning realistic grammars slow	Unclear generalization to unseen behaviors	Used a huge collection of input samples	Unclear quality of RL policy	Unclear benefit (production-optimized initial seeds)

Dynamic program analysis



Statistical error detection

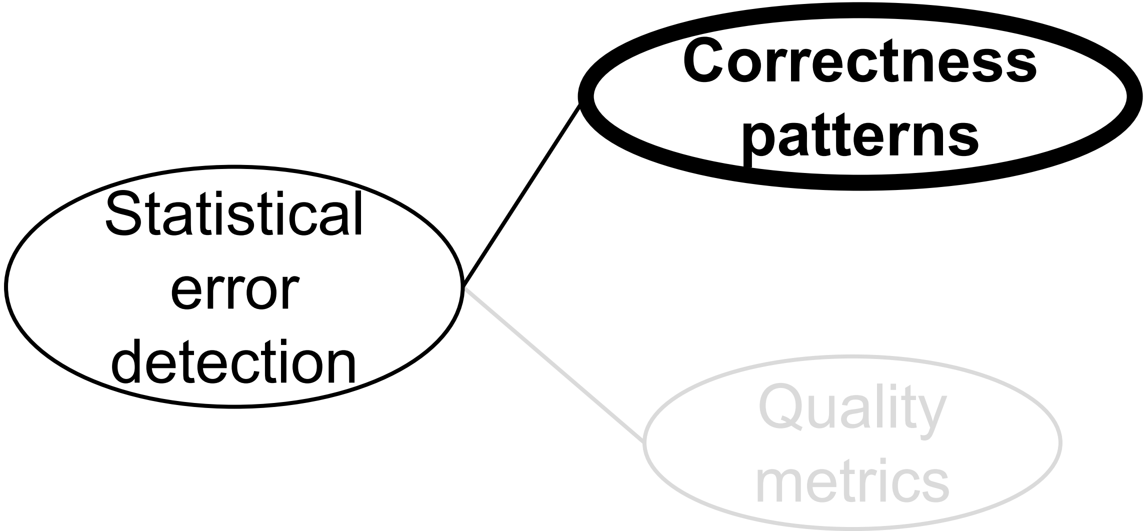


Statistical error detection

Correctness patterns

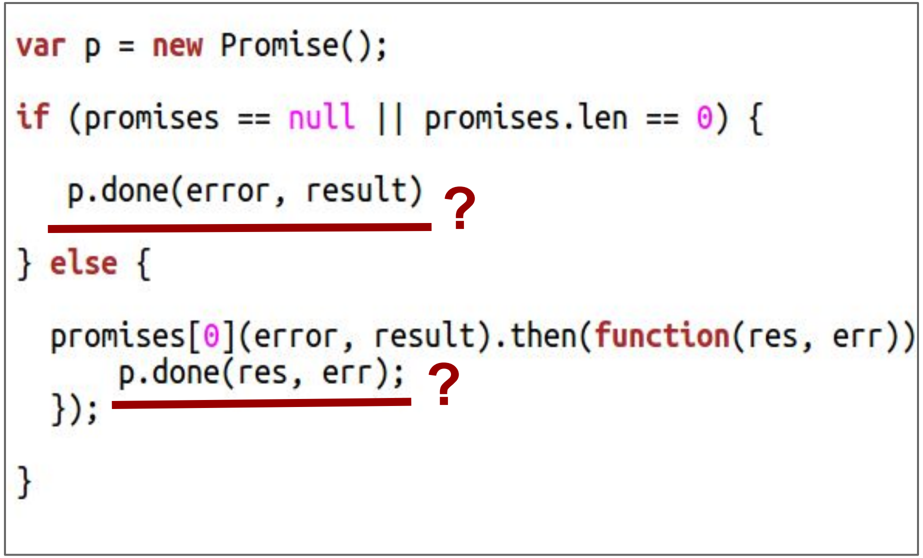
Quality metrics

Correctness patterns



Correctness patterns

- ➢ DeepBugs [31](2017)



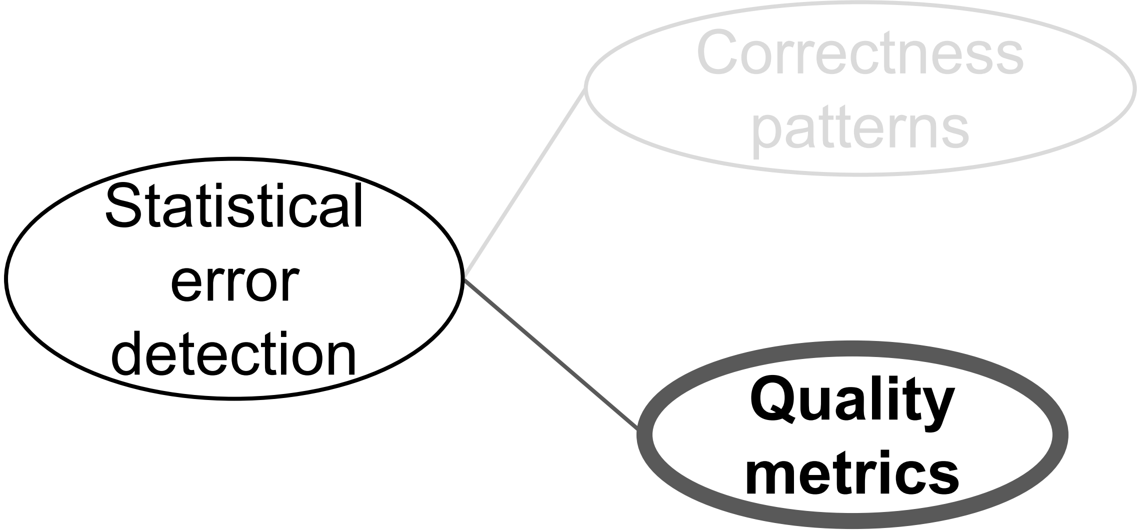
```
var p = new Promise();

if (promises == null || promises.len == 0) {

  p.done(error, result)

} else {

  promises[0](error, result).then(function(res, err))
      p.done(res, err);
  });

}
```

Key points

- Inconsistent but...which is correct?
- Most code is (hopefully) correct
- Perform transformations to create incorrect samples

➢ Need to come up with language-specific checkers

Quality metrics



Correctness patterns

Statistical error detection

Quality metrics

Quality metrics

- ➢ Transfer Defect Learning [34](2013)

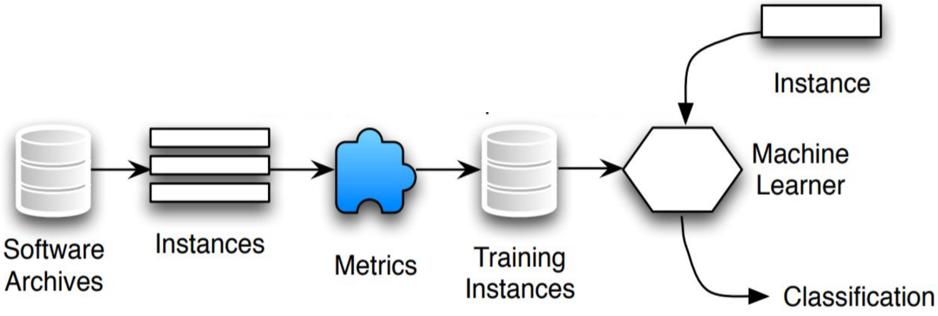


Fig. 1: Defect Prediction Process

Key points

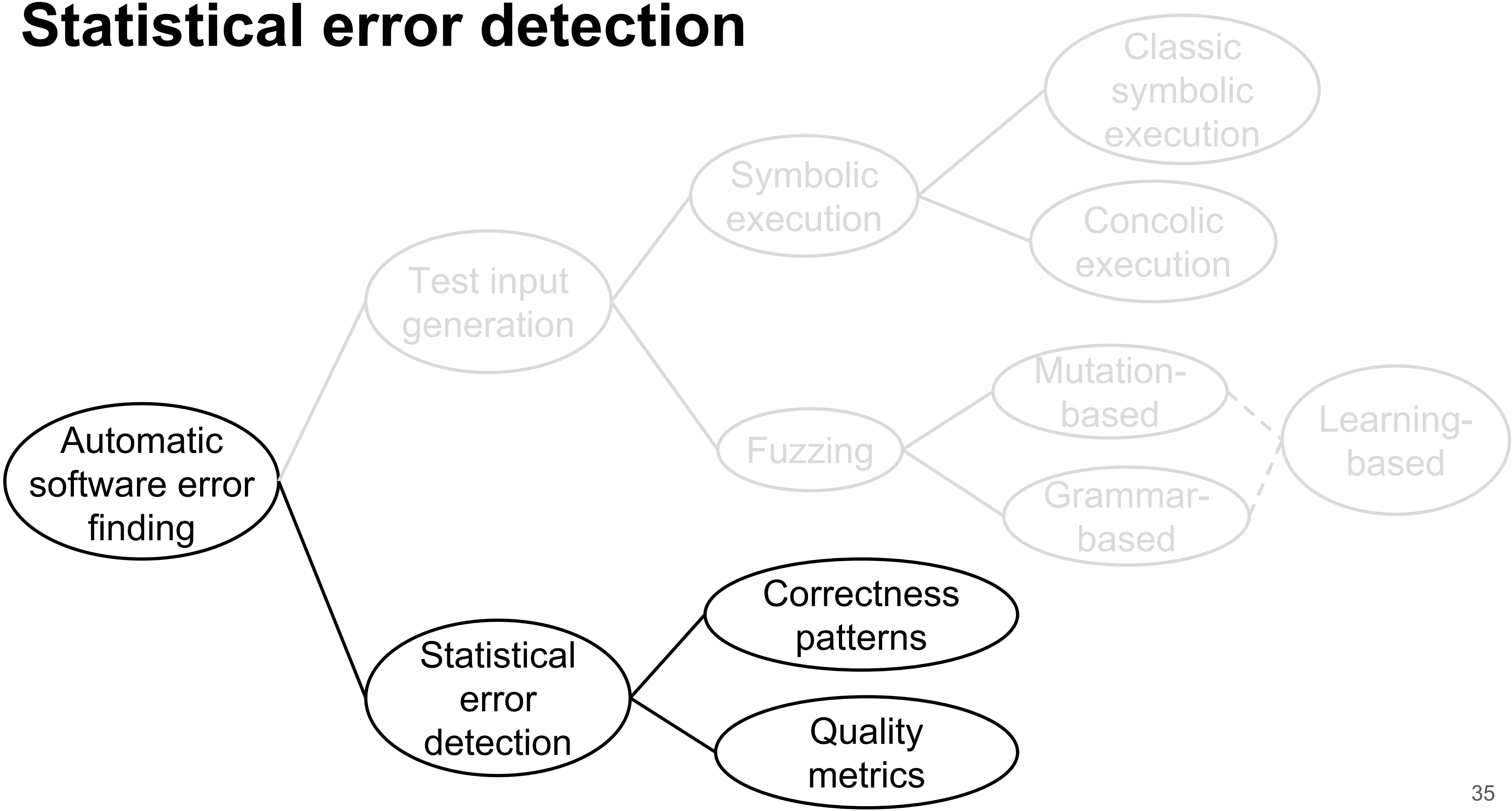
- Code quality metrics of known defects
- Predict if new files look defective
- General metrics ↔ reusable across new targets

➢ File-level reports

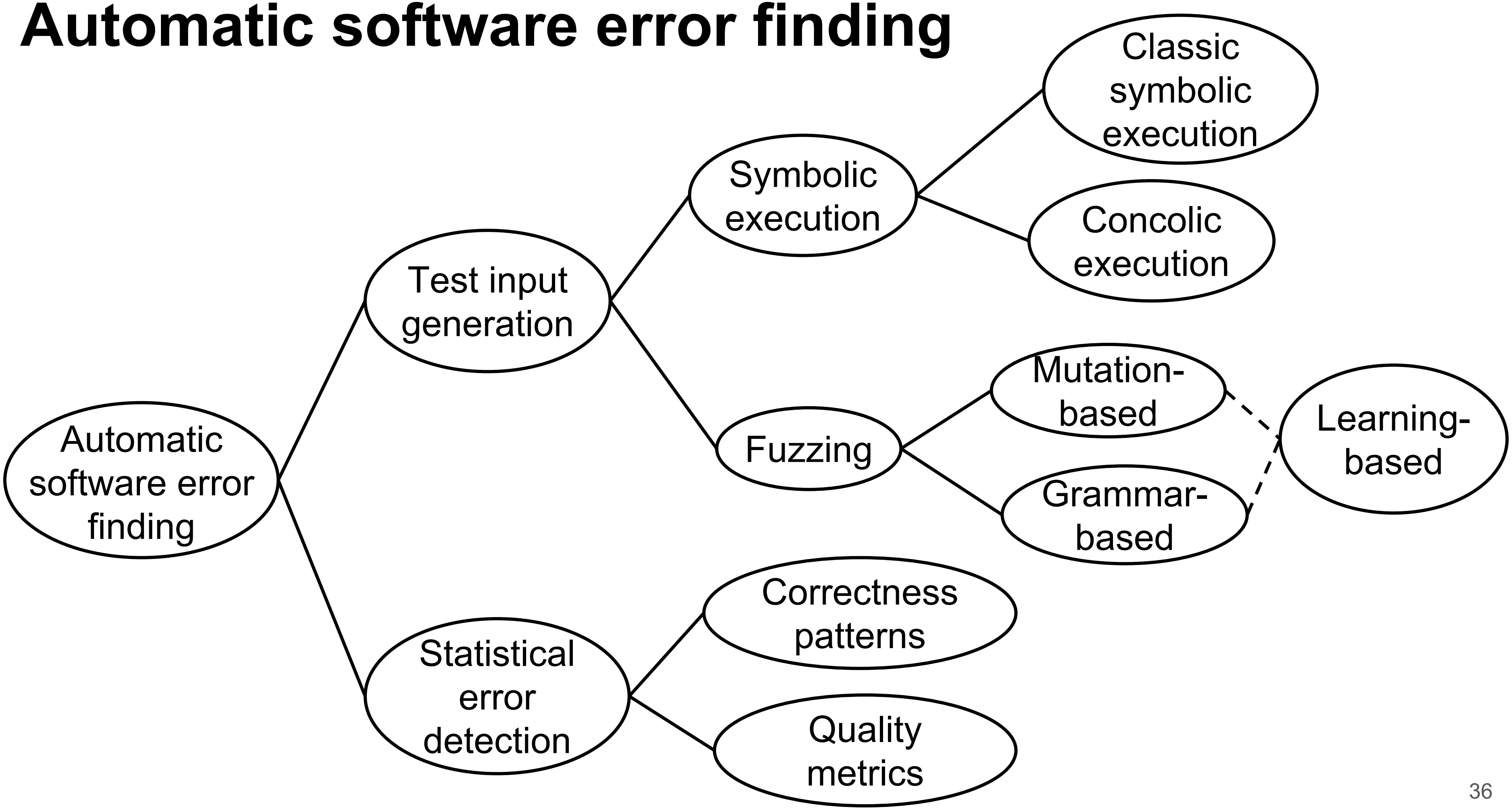
Comparative view

System	Proxy for error detection	Source-target	Transfer learning type
DeepBugs [31]	Correctness patterns	Same	N/A
Bugs as Deviant Behaviour [30]	Correctness patterns	Same	N/A
Naturalness [32]	Quality metrics	Same	N/A
TCA+ [34]	Quality metrics	Different	Domain adaptation
Semi-supervised Defect Prediction [33]	Quality metrics	Different	Inductive transfer learning

Statistical error detection



Automatic software error finding



Bibliography

Test Input Generation Using Symbolic Execution & Fuzzing

[1] King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.
[2] Cadar, Cristian, et al. "EXE: automatically generating inputs of death." ACM CCS'06
[3] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI 2008.
[4] Boonstoppel, Peter, Cristian Cadar, and Dawson Engler. "RWset: Attacking path explosion in constraint-based test generation." International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008.
[5] Tillmann, Nikolai, and Jonathan De Halleux. "Pex-white box test generation for .net." International conference on tests and proofs.Springer,Berlin,Heidelberg,2008.`
[6] Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. 'S2E: A platform for in-vivo multi-path analysis of software systems.' Acm Sigplan 2011.
[7] Bucur, Stefan, et al. "Parallel symbolic execution for automated real-world software testing." 6th conference on Computer systems. ACM, 2011.
[8] Avgerinos, Thanassis, et al. "Enhancing symbolic execution with veritesting." 36th International Conference on Software Engineering. ACM, 2014.
[9] Ramos, David A., and Dawson R. Engler. "Under-Constrained Symbolic Execution: Correctness Checking for Real Code." USENIX Security 2015.
[10] Stephens, Nick, et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." NDSS. Vol. 16. 2016.
[11] Baldoni, Roberto, et al. "A survey of symbolic execution techniques." ACM Computing Surveys (CSUR) 51.3 (2018): 50.
[12] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.
[13] Sen, Koushik, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." ACM SIGSOFT Software Engineering Notes. ACM, 2005.
[14] Godefroid, Patrice. "Compositional dynamic test generation." ACM Sigplan Notices. Vol. 42. No. 1. ACM, 2007.
[15] Majumdar, Rupak, and Koushik Sen. "Hybrid concolic testing." Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.
[16] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.
[17] Burnim, Jacob, and Koushik Sen. "Heuristics for scalable dynamic test generation." Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, 2008.
[18] Miller, Barton P., Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." Communications of the ACM (1990): 32-44.
[19] Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated whitebox fuzz testing." NDSS. Vol. 8. 2008.
[20] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." ACM Sigplan Notices. Vol. 43. No. 6. ACM, 2008.
[21] Bounimova, Ella, Patrice Godefroid, and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production. Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.
[22] McMinn, Phil. "Search-based software test data generation: a survey." Software testing, Verification and reliability 14.2 (2004): 105-156.
[23] American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/
[24] SPIKE. https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html

Test Input Generation using Symbolic Execution & Fuzzing AND Learning-based Fuzzing (dynamic)

[25] Bastani, Osbert, et al. "Synthesizing program input grammars." ACM SIGPLAN Notices. Vol. 52. No. 6. ACM, 2017.
[26] Godefroid, Patrice, Hila Peleg, and Rishabh Singh. "Learn&fuzz: Machine learning for input fuzzing." ASE 2017.
[27] Wang, Junjie, et al. "Skyfire: Data-driven seed generation for fuzzing." Security and Privacy (SP), 2017 IEEE Symposium on. IEEE, 2017.
[28] Bottinger, Konstantin, Patrice Godefroid, and Rishabh Singh. "Deep Reinforcement Fuzzing." arXiv preprint arXiv:1801.04589 (2018).
[29] She, Dongdong, et al. "NEUZZ: Efficient Fuzzing with Neural Program Learning." arXiv preprint arXiv:1807.05620 (2018).

Learning-based Program Analysis (static)

[30] Engler, Dawson, et al. "Bugs as deviant behavior: A general approach to inferring errors in systems code." SOSP 2001.
[31] Pradel, Michael, and Koushik Sen. "Deep Learning to Find Bugs." (2017).
[32] Ray, Baishakhi, et al. "On the" naturalness" of buggy code." Software Engineering (ICSE), 2016. (use of statistical natural language processing models)

Learning-based Program Analysis (static) AND Transfer Learning

[33] Lu, Huihua, Bojan Cukic, and Mark Culp. "Software defect prediction using semi-supervised learning with dimension reduction." Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, 2012.
[34] Nam, Jaechang, Sinno Jialin Pan, and Sunghun Kim. "Transfer defect learning." Software Engineering (ICSE), 2013.

Transfer Learning

[35] Pan, Sinno Jialin, and Qiang Yang. "A survey on transfer learning." IEEE Transactions on knowledge and data engineering 22.10 (2010): 1345-1359.
[36] Ruder, Sebastian. "Transfer learning: Machine learning's next frontier." (2017).
[37] Caruana, Rich. "Multitask learning." Machine learning 28.1 (1997): 41-75.
[38] Yosinski, Jason, et al. "How transferable are features in deep neural networks?." Advances in neural information processing systems. 2014.

Software Model Checking (not discussed in this presentation)

- Godefroid, Patrice."Model checking for programming languages using VeriSoft." Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1997.
- Musuvathi, Madanlal, et al. "CMC: A pragmatic approach to model checking real code." ACM SIGOPS Operating Systems Review 36.SI (2002): 75-88.
- Visser, Willem, Corina S. Pasareanu, and Sarfraz Khurshid. "Test input generation with Java PathFinder." ACM SIGSOFT Software Engineering Notes 29.4 (2004): 97-107.
- Visser, Willem, et al. "Model checking programs." Automated software engineering 10.2 (2003): 203-232.
- Yang, Junfeng, Can Sar, and Dawson Engler. "Explode: a lightweight, general system for finding serious storage system errors." Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.