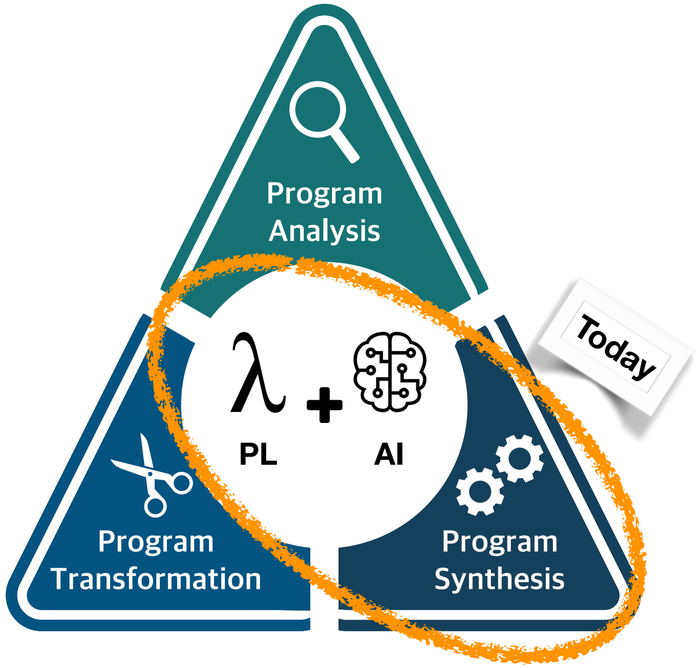
IS593: Language-based Security

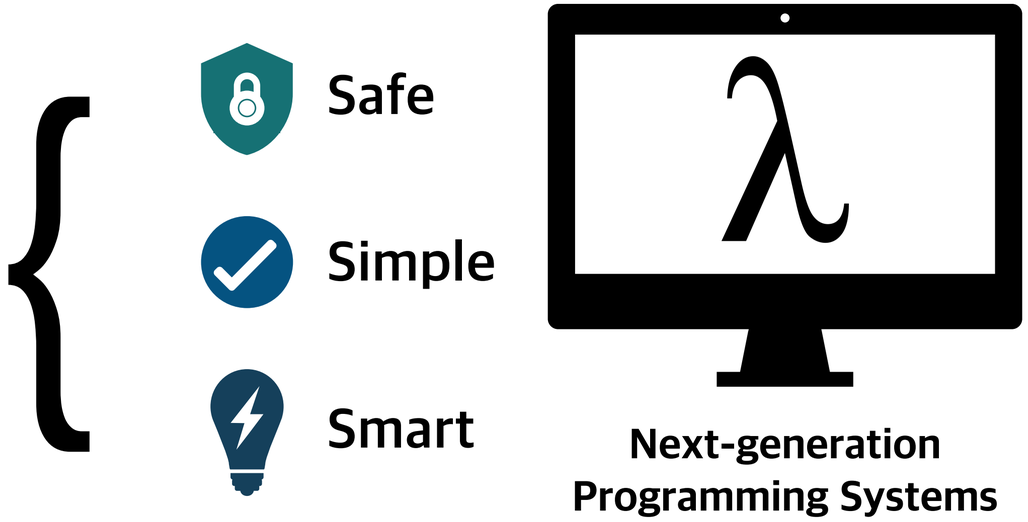
14. Program Synthesis

Kihong Heo

KAIST

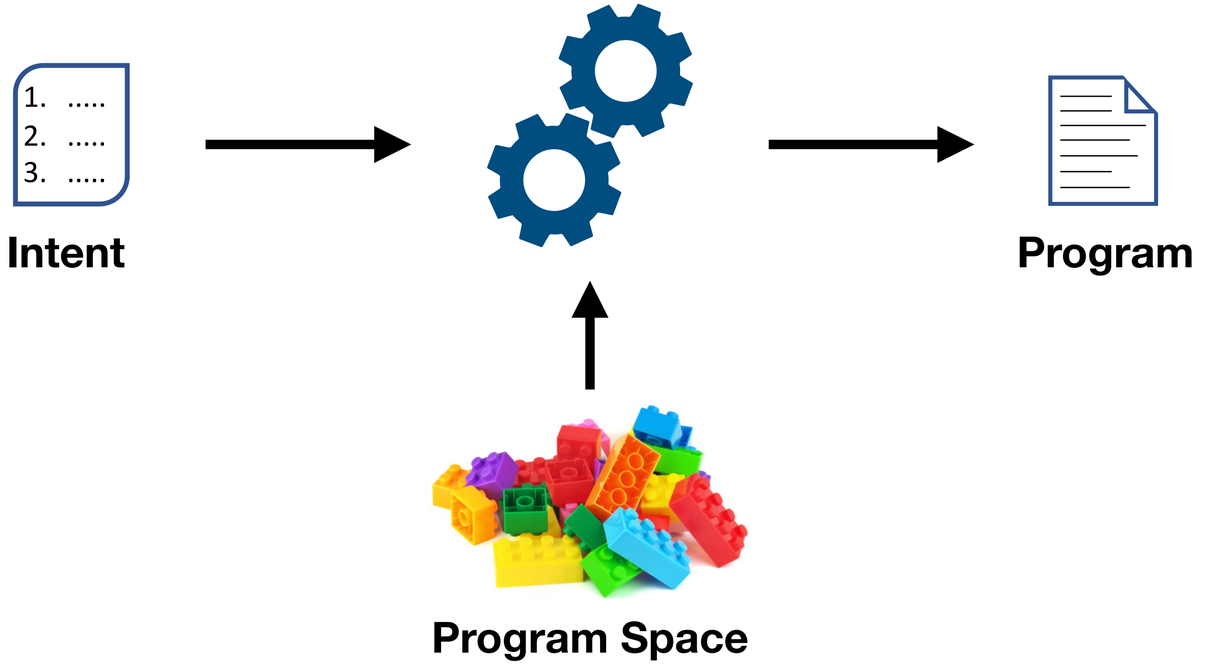
New Waves in Language-based Security



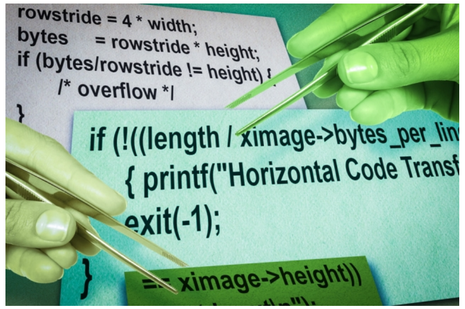


Program Synthesis

- A task of automatically finding programs
  - that satisfy user intent from the underlying program space



Applications



Automatic Program Repair

	A	B	C
1	**Full Name**	**First Name**	**Last Name**
2	Kihong Heo	Kihong	Heo
3	Michael Jordan	Michael	
4	Thierry Henry	Thierry	
5			
6			

End-user Programming

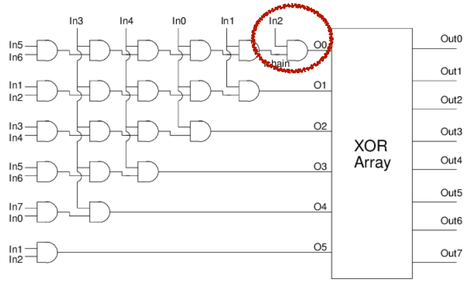
SW Education

```
// original code
average (bv32 x, bv32 y) {
  bv64 x1 = x;
  bv64 y1 = y;
  bv64 z1 = (x1+y1)/2;
  bv32 z = z1;
  return z;
}
```

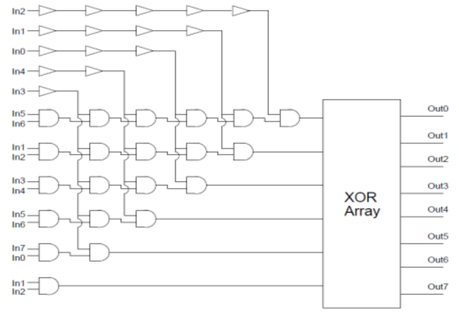
```
// optimized code without
extension to 64 bit vectors
average (bv32 x, bv32 y) {
  return (x and y)
    + ((x xor y) >> 1);
}
```

Superoptimization (performance)*

Original circuit



Side-channel Attack Resilient Version



Superoptimization (security)*

*Examples from Rajeev Alur's slides

Dimensions in Program Synthesis

- User intent (semantic specification)
  - E.g., logical formula, examples, natural languages, etc
- Search space (syntactic specification)
  - E.g., existing languages, specially designed DSL, programs with holes, etc
- Search technique
  - E.g., enumerative, stochastic, constraint-based, etc

***This lecture**

Inductive Synthesis

- Given a set of examples, find a program consistent with the examples
  - “Programming by example (PBE)”

x	f(x)
1	1
2	3
3	5
4	7

- Compared to conventional machine learning?
  - Interpretable, verifiable, and expressive

Program Space

- Should strike a good balance between **expressiveness** and **efficiency**
- Usually a restricted form of context-free grammar
  - E.g., restrictions on operators or control structures
- In this lecture, consider all programs from the following grammar:

$$S \to x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S$$
$$B \to S \leq S \mid S = S$$

Enumerative Search

- **Explicitly enumerate** programs in the search space until finding a solution
- How to enumerate?
  - Top-down: starting from the start non-terminal
  - Bottom-up: starting from terminals
- Effective search space pruning techniques are needed

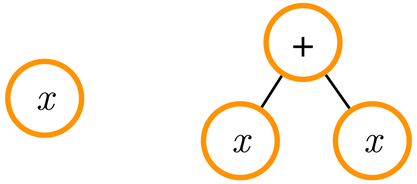
Size of the Problem Space

$$S \rightarrow x \mid S + S$$

depth ≤ 1

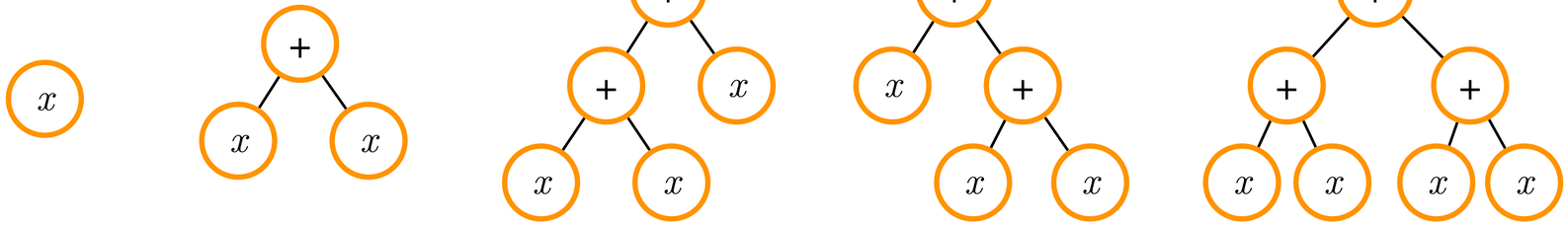
N(1) = 1

depth ≤ 2



N(2) = 2

depth ≤ 3



N(3) = 5

N(d) = 1 + N(d - 1)²

*Examples from Nadia Polikarpova's slides

How Big is the Space?

$$S \rightarrow x \mid S + S$$

N(d) = 1 + N(d - 1)²

N(1) = 1
N(2) = 2
N(3) = 5
N(4) = 26
N(5) = 677
N(6) = 458330
N(7) = 210066388901
N(8) = 44127887745906175987802
N(9) = 19472704769152964495597034454938489304527912 05
N(10) = 37918623102659260828682350280278932773702331522473885847617341507177682544103411753253520 26

!!

*Examples from Nadia Polikarpova's slides

Top-down Enumeration

- Start from the **start non-terminal**
- **Expand** remaining non-terminals using production rules
- Example:

Find a function $f(x, y)$ where $f(0, 1) = 1 \wedge f(1, 2) = 3$

Grammar

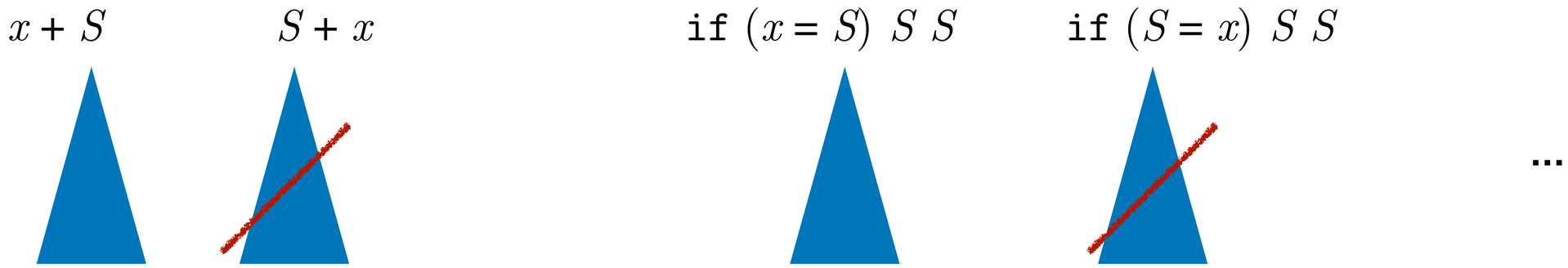
$$S \to x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S$$
$$B \to S \leq S \mid S = S$$

Enumeration

iter 0	S					
iter 1	x	y	S + S	S - S	if B S S	
iter 2	x + S	y + S	x - S	y - S	if $(S \leq S)$ S S	...
iter 3	x + x	x + y	y + x	y - y	if $(x \leq S)$ S S	...

Search Space Pruning

- Discard **redundant** or **unpromising** subprograms as early as possible
  - Estimating program's behavior beforehand: in general static analysis problem
- In this lecture, simple equivalence checking heuristics



Top-down Enumeration Algorithm

$\textbf{\textit{TopDown}}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow \{S\}$

`while` $P \neq \emptyset:$

$p \leftarrow Dequeue(P)$

`if` $\phi(p)$: `return` p

$P' \leftarrow Unroll(G, p)$

`forall` $p' \in P':$

`if` $\neg Subsumed(P, p'):$

$P \leftarrow Enqueue(P, p')$

$\textbf{\textit{Unroll}}(\text{grammar } G, \text{program } p):$

$P' \leftarrow \emptyset$

`forall` $A \in p:$

`forall` $(A \rightarrow B) \in G:$

$p' \leftarrow p[B/A]$

$P' \leftarrow P' \cup \{p'\}$

`return` P'

Bottom-up Enumeration

- Start from **terminals**
- **Combine** sub-programs into larger ones using production rules
- Example:

Find a function $f(x, y)$ where $f(3, 1) = 3 \wedge f(1, 2) = 2$

Grammar

$$S \rightarrow x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S$$
$$B \rightarrow S \leq S \mid S = S$$

Enumeration

iter 0	x	y			
iter 1	x + y	x - y	$x \leq y$	$x = y$	
iter 2	$x + x + y$	$x + x - y$	...	$\texttt{if}\ (x \leq y)\ y\ x$	...
iter 3	$x + x + x + y$		...	$\texttt{if}\ (x \leq y)\ (y + x)\ x$	

Search Space Pruning

- Discard **redundant** or **unpromising** subprograms as early as possible
- Problem: candidates are all ground but might not be whole
- Solution for PBE: **observation equivalence**
  - We only care of equivalence on the given inputs
- For example, the programs below are observationally equivalent modulo the inputs (1,0) and (2,1)

$$x \qquad\qquad \texttt{if}\ (x \leq y)\ y\ x$$

Bottom-up Enumeration

$\textbf{\textit{BottomUp}}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow$ set of all terminals in G

`while true :`

`forall` $p \in P'$:

`if` $\phi(p)$: `return` p

$P' \leftarrow Grow(G, P)$

$P \leftarrow P \cup \{p' \in P' \mid$ `forall` $p \in P : \neg Equiv(\phi, p, p')\}$

$\textbf{\textit{Grow}}(\text{grammar } G, \text{set of programs } P):$

$P' \leftarrow \emptyset$

`forall` $(A \to B) \in G$:

$P' = P' \cup \{B[p/C] \mid p \in P, C \to^* p\}$

`return` P'

Problem

- Blindly search over the large search space without any guidance
- #programs grows exponentially in program size

iter 0	x	y				
iter 1	$x + x$	$x - x$	$x + y$	...	$x \leq y$	...
iter 2	$x + x + y$	$x + x - y$	...	if $(x \leq y)$ y x		...
iter 3	$x + x + x + y$		...	if $(x \leq y)$ $(y + x)$ x		

But which one is more likely to be a solution?

$x - x$ **vs.** if $(x \leq y)$ y x

Statistical Regularities in Programs

- Programs often contain **repetitive** and **predictable** patterns

```
for (i = 0; i < 100; ??)
```

- **Statistical** program models: probabilistic distribution over programs
  - E.g., n-gram, probabilistic context-free gramma (PCFG), etc

$$Pr(?? \rightarrow \texttt{i++} \mid \texttt{for (i = 0; i < 100; ??)}) = 0.85$$
$$Pr(?? \rightarrow \texttt{i--} \mid \texttt{for (i = 0; i < 100; ??)}) = 0.01$$

- Applications: code completion, deobfuscation, program repair, etc

A Solution: Euphony*

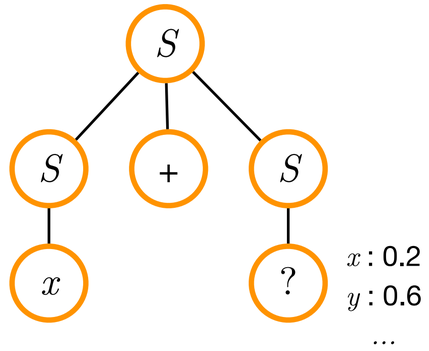
- Enumerate programs by **likelihood**, not by program size
- Likelihood is provided by **probabilistic models** over programs
  - "*How likely is the candidate program?*"
- Try the **most likely** (highest probability) candidate first

*Lee et al., Accelerating Search-Based Synthesis Using Learned Probabilistic Models, PLDI, 2018

Probabilistic Model

- Learn a probabilistic model of programs from a corpus of programs
  - Human-written or auto-generated programs by other synthesizers
- A wide range of models is applicable

$Pr(S \rightarrow S + S) = 0.3$
$Pr(S \rightarrow x \mid S + S) = 0.8$
...
$Pr(S \rightarrow x \mid x + S) = 0.2$
$Pr(S \rightarrow y \mid x + S) = 0.6$
...



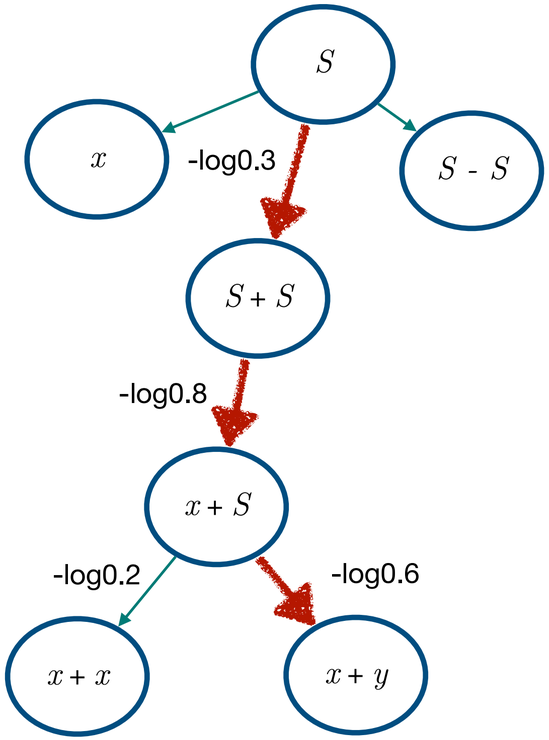
Program Corpus

Learned Probabilistic Model

Probability of Programs

Guided Enumeration by Probabilistic Model

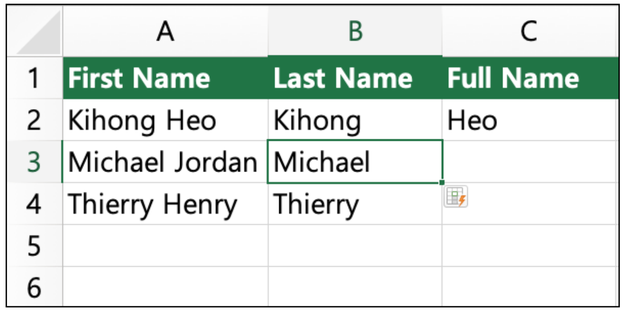
- Given a model, construct a directed graph
  - Node: sentential forms
  - Weight: negative log probability of a production rule
- Compute the shortest path
  - starting from the start symbol to the program
  - E.g., Dijkstra's, A*, etc



$Pr(S \rightarrow S + S) = 0.3$
$Pr(S \rightarrow x \mid S + S) = 0.8$
...
$Pr(S \rightarrow x \mid x + S) = 0.2$
$Pr(S \rightarrow y \mid x + S) = 0.6$
...

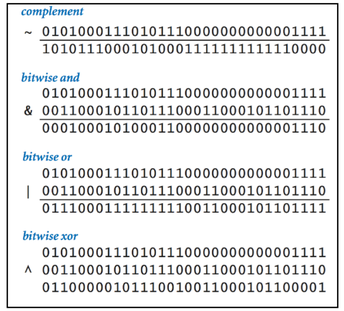
Experimental Setup

- 1167 tasks from 3 different domains

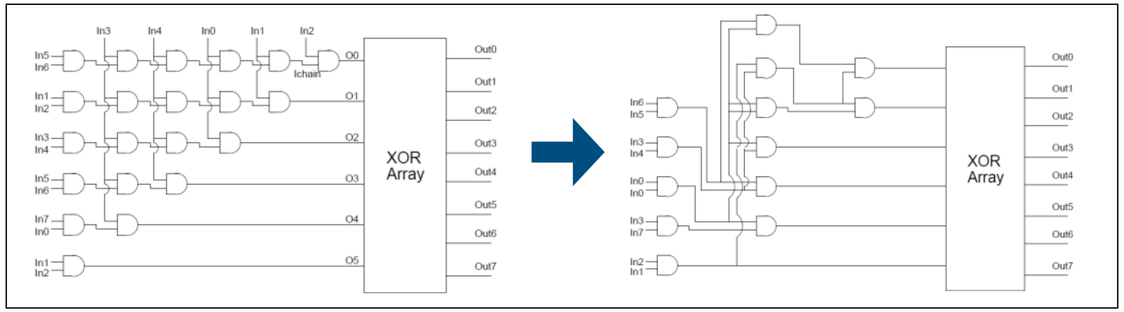


	A	B	C
1	First Name	Last Name	Full Name
2	Kihong Heo	Kihong	Heo
3	Michael Jordan	Michael	
4	Thierry Henry	Thierry	
5			
6			

STRING: End-user programming for string manipulations
(205 tasks)



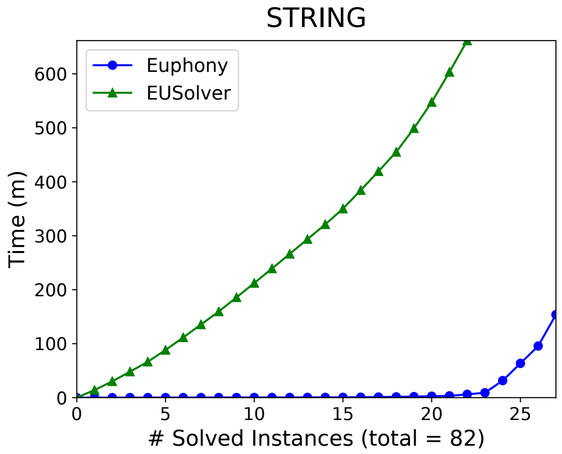
BITVEC: Efficient low-level algorithms
(750 tasks)

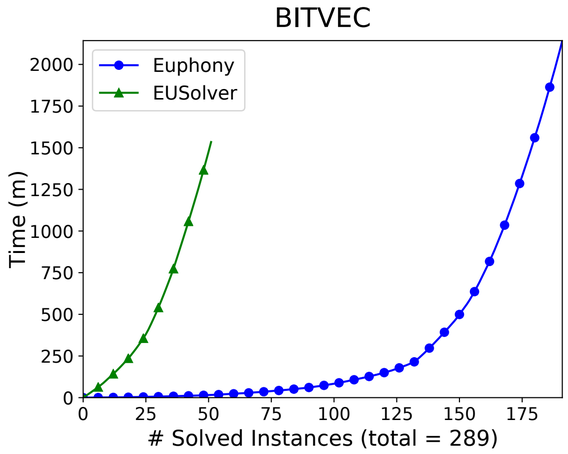


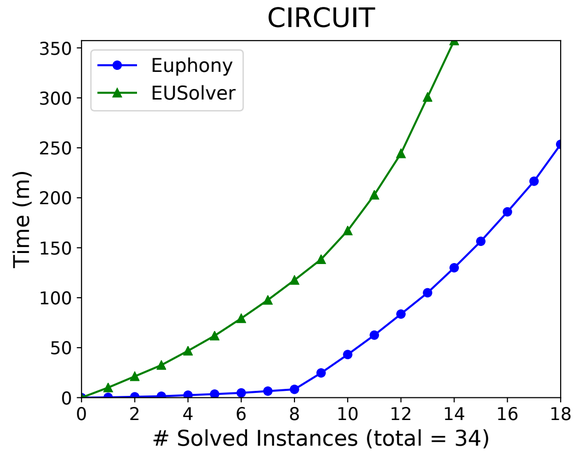
CIRCUIT: Attack-resistant crypto circuits generations
(212 tasks)

Effectiveness

- Comparison to EUSolver (a program synthesizer without prob. guidance)
  - Training: 762 tasks solved by EUSolver in 10 minutes
  - Testing: 405 (timeout: 1 hour)







Summary

- Program synthesis: **automated programming** systems
- Challenge: **huge search space**
- Euphony: a program synthesizer guided by **learned probabilistic model**
  - E.g., probabilistic program model + shortest path finding
- Need a lot more research on efficient search
  - E.g., advanced learning techniques, static analysis, constraint solving, etc