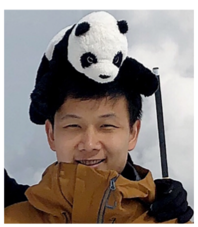
DeBIN: Predicting Debug Information in Stripped Binaries

https://debin.ai



Jingxuan He

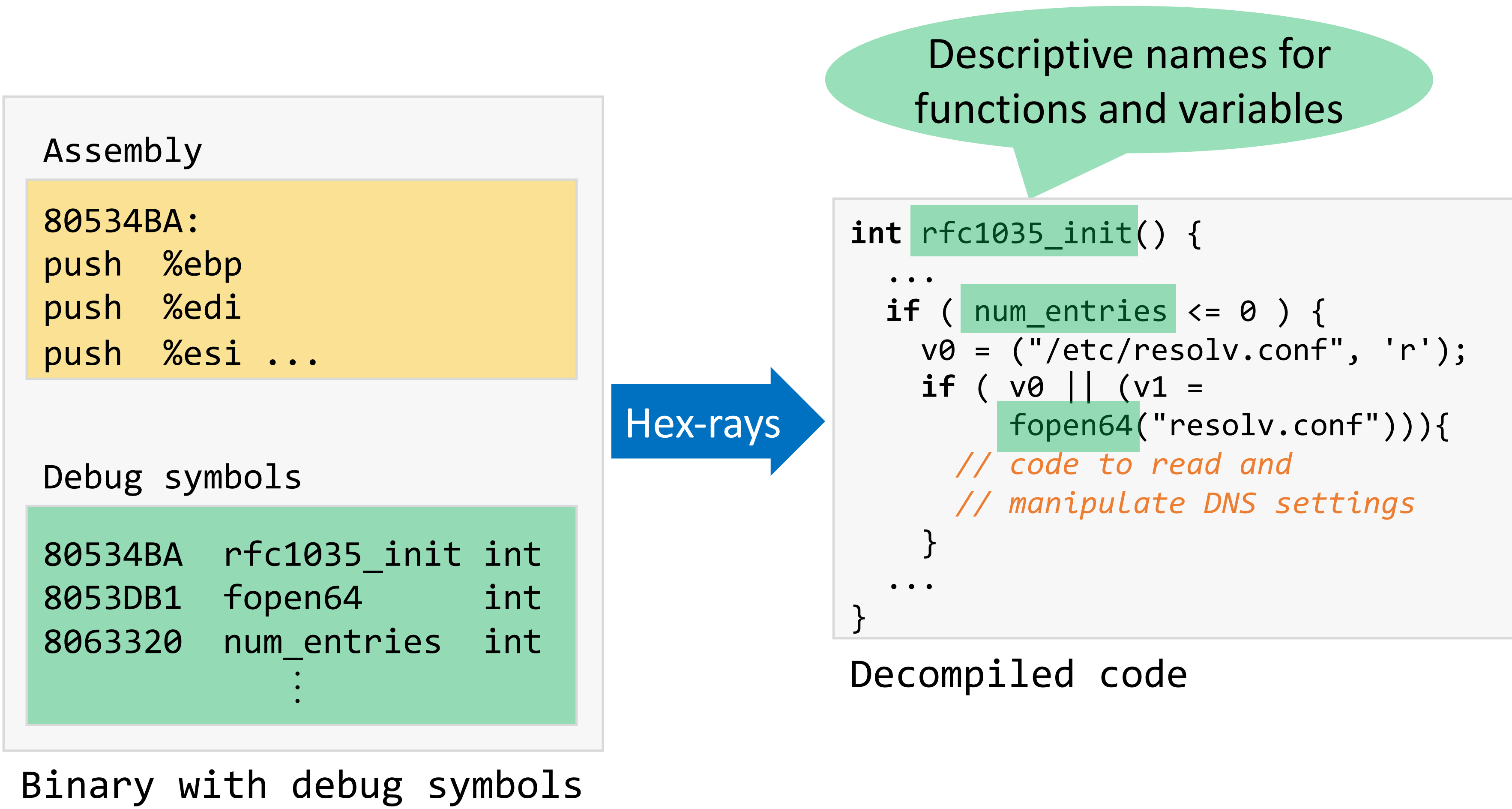
Pesho Ivanov

Petar Tsankov

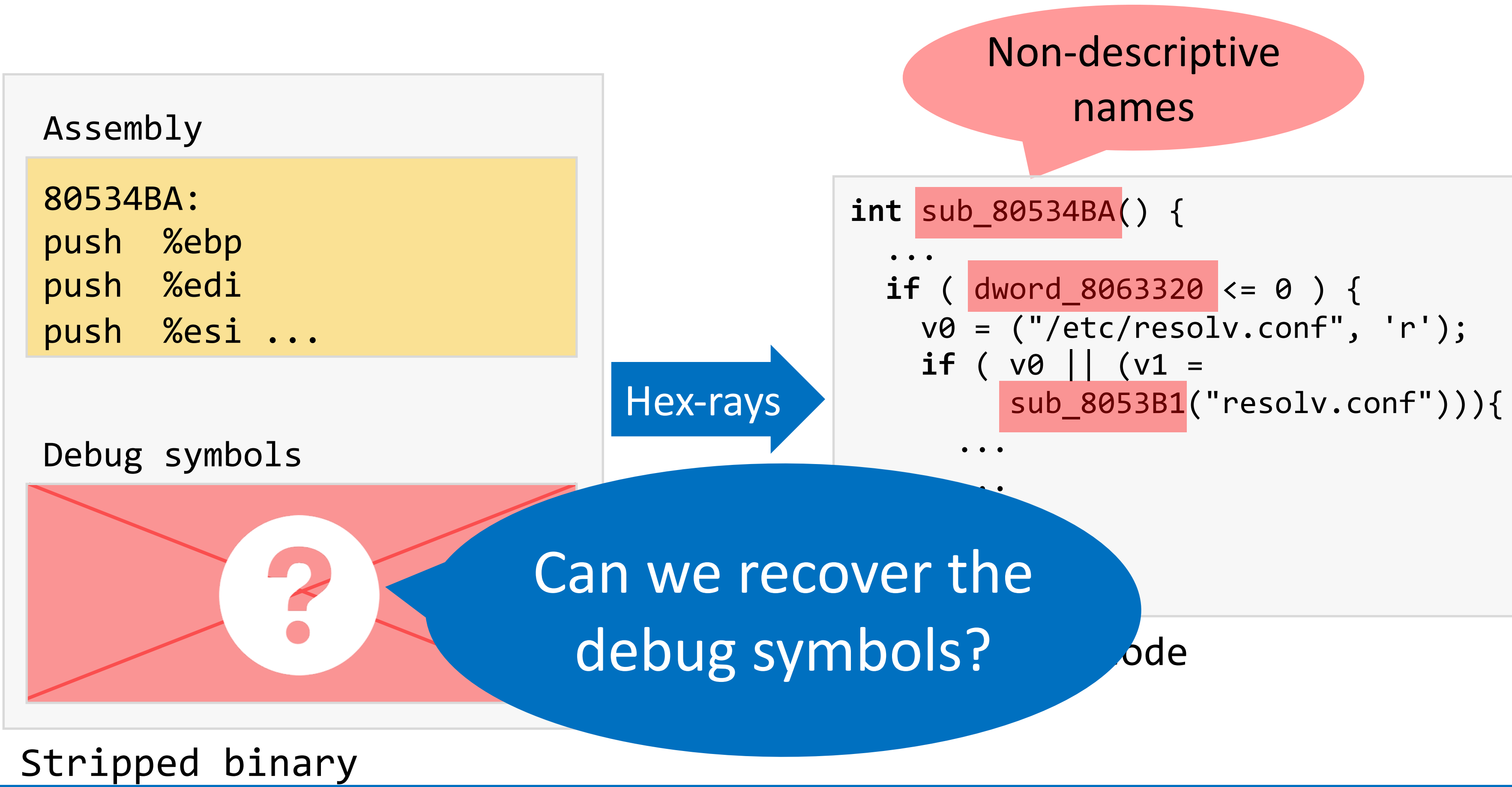
Veselin Raychev

Martin Vechev

Binaries with debug symbols



Stripped binaries



Yes, with roughly 65% accuracy!

Challenges

Computes
1 + 2 + … + n

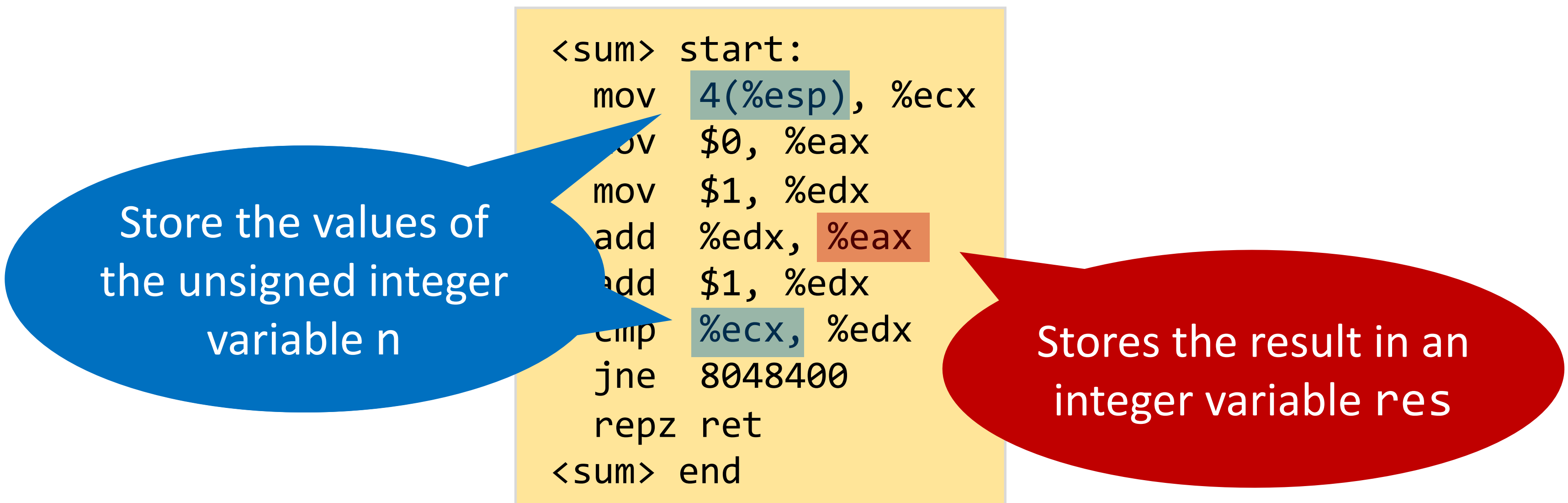
Stores the value of
a semantic variable

```
<sum> start:
   mov   4(%esp), %ecx
   mov   $0, %eax
   mov   $1, %edx
   add   %edx, %eax
   add   $1, %edx
   cmp   %ecx, %edx
   jne   8048400
   repz ret
<sum> end
```

Stores intermediate
(non-semantic) value

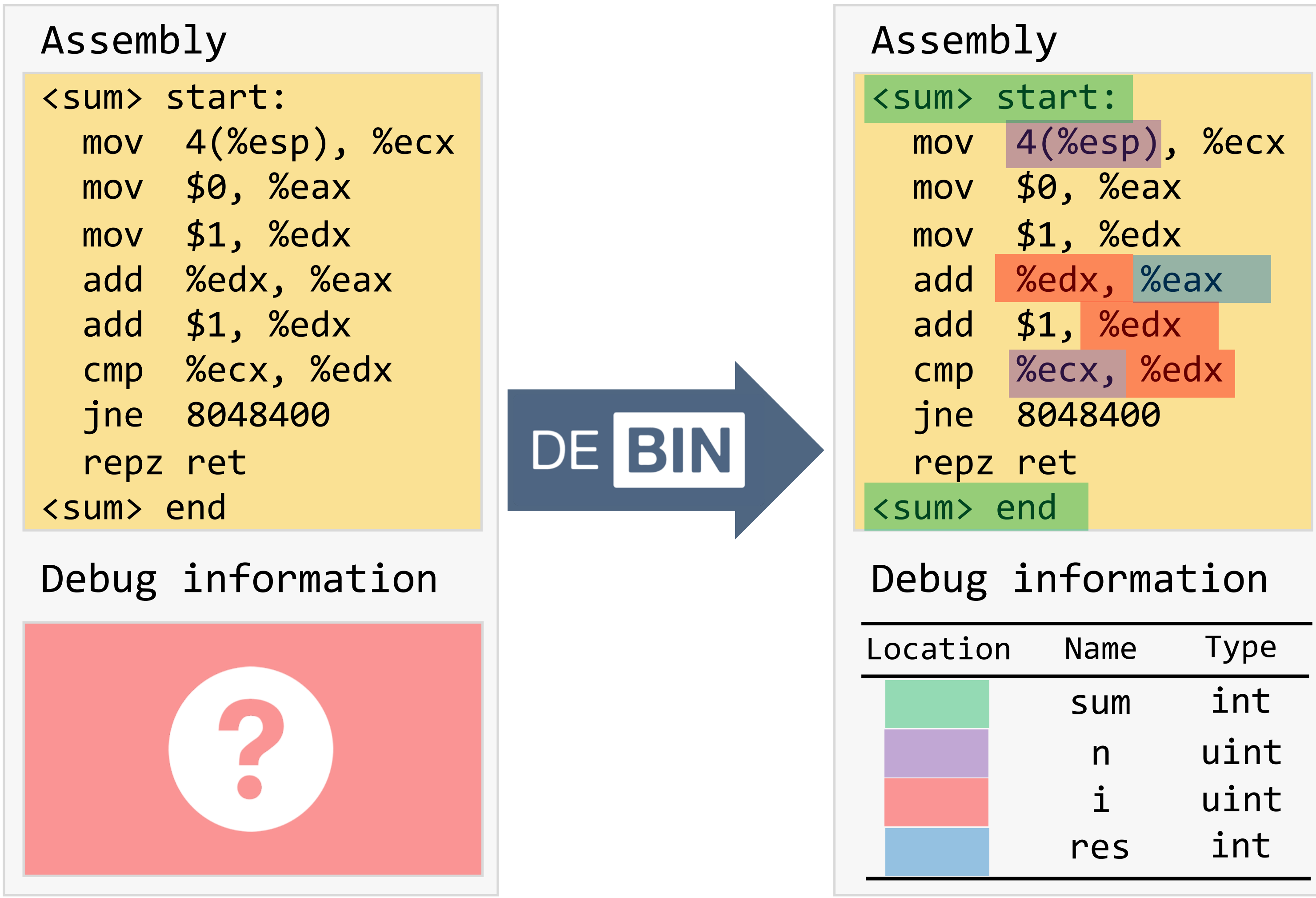
1. No mapping from registers and memory offsets to semantic variables

Challenges

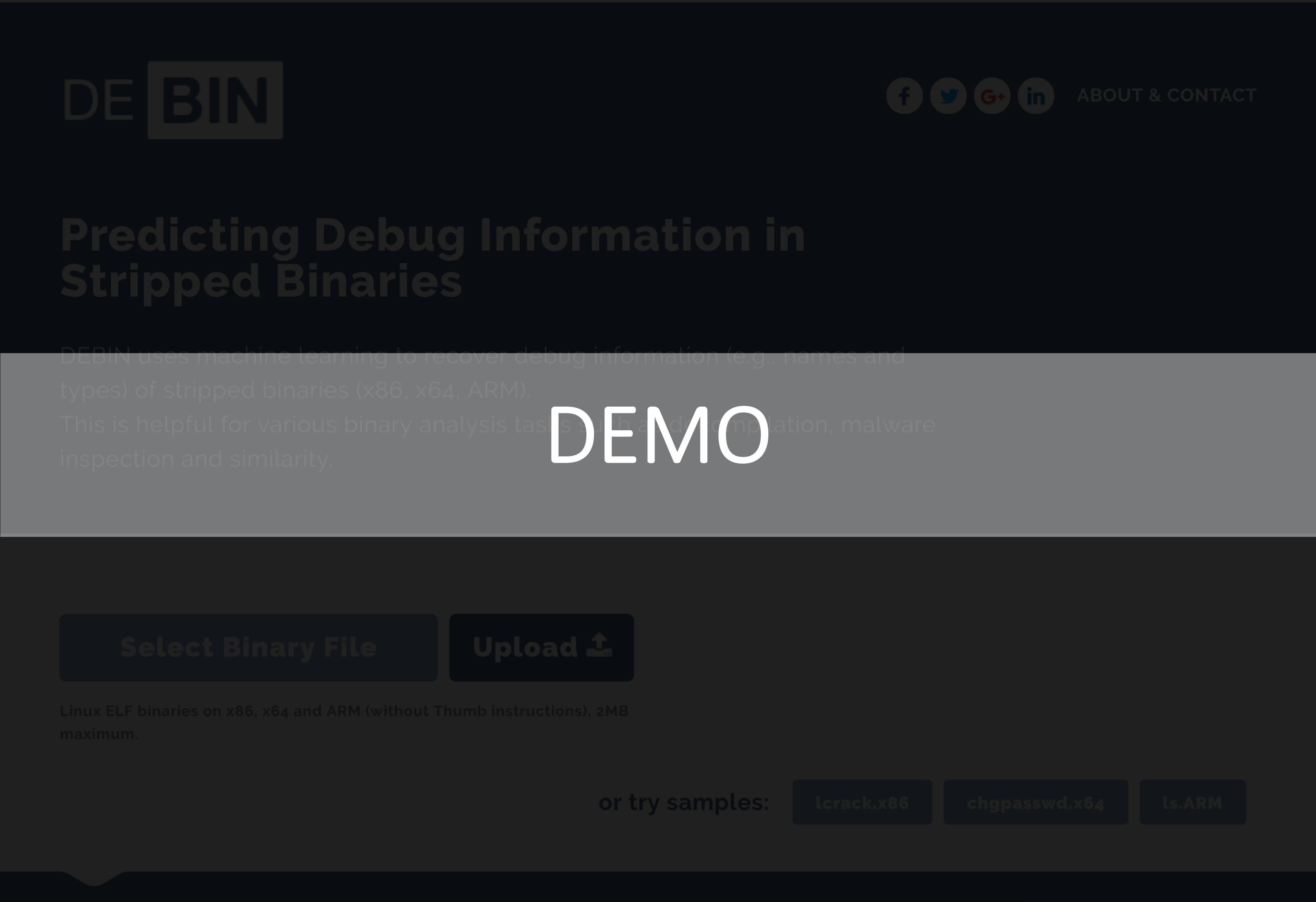


1. No mapping from registers and memory offsets to semantic variables
2. No names and types

DeBIN: Recovering debug information

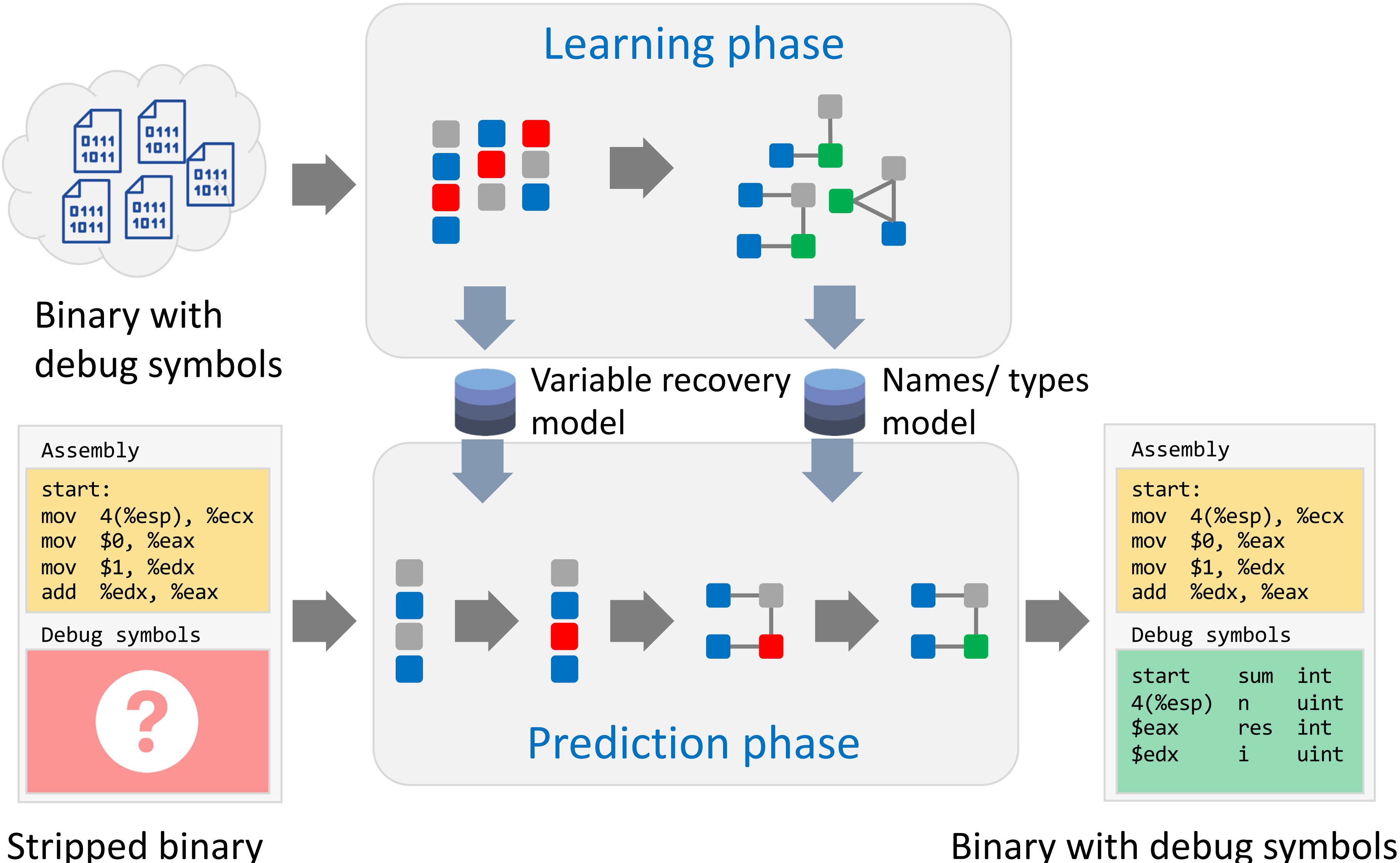


DeBIN recovers location information, types, and names


DEMO
DE BIN
ABOUT & CONTACT
Predicting Debug Information in Stripped Binaries
DEBIN uses machine learning to recover debug information (e.g., names and types) of stripped binaries (x86, x64, ARM).
Select Binary File
Upload
Linux ELF binaries on x86, x64 and ARM (without Thumb instructions), 2MB maximum.
or try samples:
lcrack.x86
chgpasswd.x64
ls.ARM

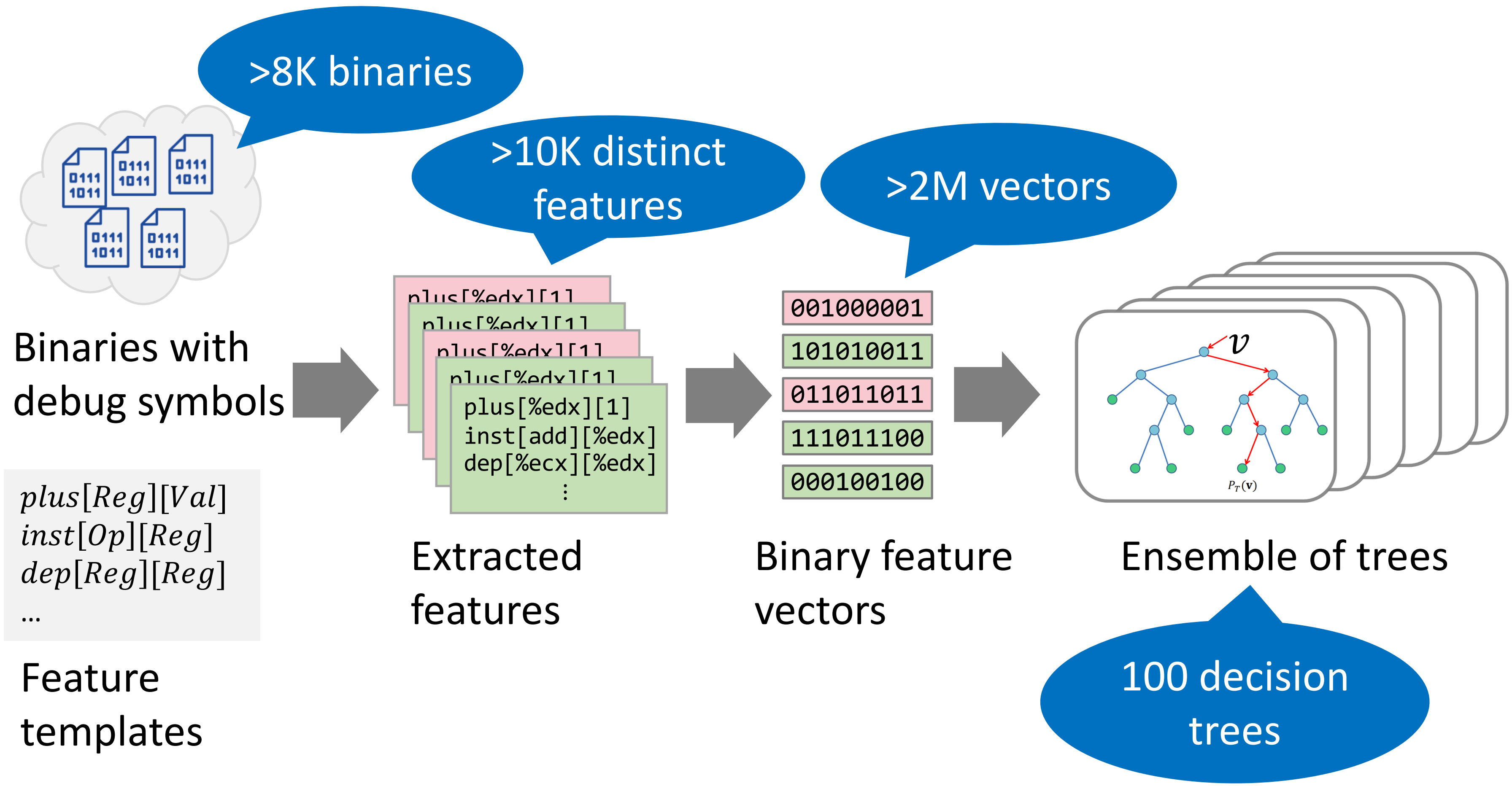
How does DeBIN work?

DeBIN: System overview



Step 1: Recovering variables

Learning how to recover variables



Variable recovery

Assembly

```
mov   4(%esp), %ecx
mov   $0, %eax
mov   $1, %edx
add   %edx, %eax
add   $1, %edx.2
cmp   %ecx, %edx
jne   8048400
repz ret
```

`%edx.2`

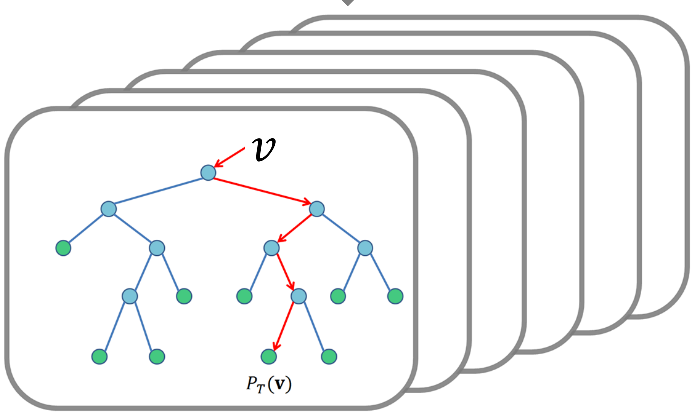
Register

```
plus[%edx][1]
inst[add][%edx]
⋮
```

Features

`00100101010001`

Feature vector v



Extremely randomized trees

sem
(DeBIN will predict name and type)

tmp
(stores an intermediate value)

Extremely randomized trees, Pierre Geurts, Damien Ernst, and Louis Wehenkel, Machine Learning 2006

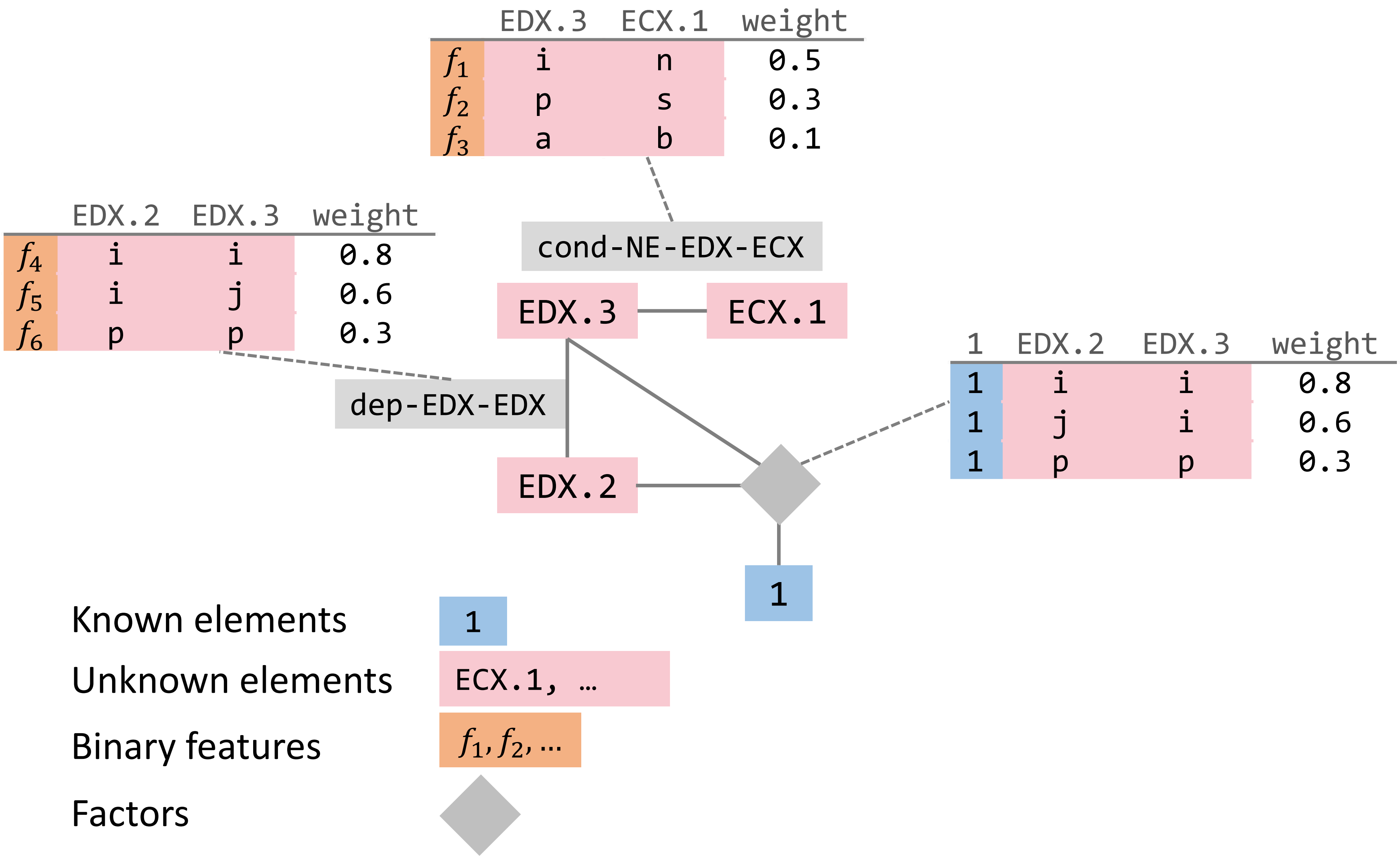
Step 2: Predicting names and types

Probabilistic graphical model

	EDX.3	ECX.1	weight
f_1	i	n	0.5
f_2	p	s	0.3
f_3	a	b	0.1

	EDX.2	EDX.3	weight
f_4	i	i	0.8
f_5	i	j	0.6
f_6	p	p	0.3

1	EDX.2	EDX.3	weight
1	i	i	0.8
1	j	i	0.6
1	p	p	0.3



Probabilistic graphical model

Relationship	Template	Condition for adding an edge
		Function Relationships
Element used in Function	(f, v, `func-`**`loc`**(v))	variable v is accessed inside the scope of function f
	(f, a, `arg-`**`loc`**(a))	variable a is an argument of function f by calling conventions
	(f, c, `func-str`)	string constant c is accessed inside the scope of function f
	(f, s, `func-stack`)	stack location s is allocated for function f
Function Call	(f_1, f_2, `call`)	function f_2 is called by function f_1
		Variable Relationships
Instruction	(v, *insn*, `insn-`**`loc`**(v))	there is an instruction *insn* (e.g., `add`) that operates on variable v
Location	(v, l, `locates-at`)	variable v locates at location l (e.g., memory offset `mem[RSP+16]`)
Locality	(v_1, v_2, `local-`**`loc`**(v_1))	variable v_1 and v_2 are locally allocated (e.g., `EDX.2` and `EDX.3`)
Dependency	(v_1, v_2, `dep-`**`loc`**(v_1)`-`**`loc`**(v_2))	variable v_1 is dependent on variable v_2
Operation	(v, *op*, `unary-`**`loc`**(v))	unary operation *op* (e.g. unsigned and low cast) on variable v
	(n_1, n_2, *op*`-`**`loc`**(n_1)`-`**`loc`**(n_2))	binary operation *op* (e.g., +, left shift « and etc.) on node n_1 and n_2
	(v_1, v_2, `phi-`**`loc`**(v_1))	there is a ϕ expression in BAP-IR: $v_1 = \phi(\ldots v_2, \ldots)$
Conditional	(v, *op*, `cond-unary`)	there is a conditional expression *op* (v) (e.g., `not (EAX.2)`)
	(n_1, n_2, `cond-`*op*`-`**`loc`**(n_1)`-`**`loc`**(n_2))	there is a conditional expression n_1 *op* n_2 (e.g. `EDX.3!=ECX.1`)
Argument	(f, a, `call-arg-`**`loc`**(a))	there is a call $f(\ldots, a, \ldots)$ with argument a
		Type Relationships
Operation	(t, *op*, `t-unary-`**`loc`**(t))	unary operation *op* on type t
	(t_1, t_2, `t-`*op*`-`**`loc`**(t_1)`-`**`loc`**(t_2))	binary operation *op* on type t_1 and t_2
	(t_1, t_2, `t-phi-`**`loc`**(t_1))	there is a ϕ expression: $t_1 = \phi(\ldots t_2, \ldots)$
Conditional	(t, *op*, `t-cond-unary`)	there is a unary conditional expression *op* (t)
	(t_1, t_2, `t-cond-`*op*`-`**`loc`**(t_1)`-`**`loc`**(t_2)))	there is a binary conditional expression t_1 *op* t_2
Argument	(f, t, `t-call-arg-`**`loc`**(t))	call $f(\ldots, t, \ldots)$ with an argument of type t
Name & Type	(v, t, `type-`**`loc`**(v))	variable v is of type t
	(f, t, `func-type`)	function f is of type t

EDX.2

f_4	i
f_5	i
f_6	p

ght

.8

.6

.3

Know

Unkn

Binar

Factors

Probabilistic graphical model

	EDX.3	ECX.1	weight
f_1	i	n	0.5
f_2	p	s	0.3
f_3	a	b	0.1

	EDX.2	EDX.3	weight
f_4	i	i	0.8
f_5	i	j	0.6
f_6	p	p	0.3

dep-EDX

EDX.3	weight
i	0.8
i	0.6
p	0.3

Next

How are the features and their weights learned?

1

Known elements 1

Unknown elements ECX.1, …

Binary features $f_1, f_2, \ldots$

Factors

Learning how to predict names and types

> 8,000 binaries

Binaries with debug symbols

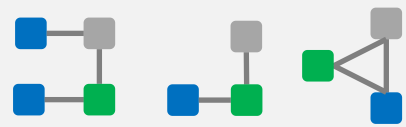
Static analysis

(f_{unary}, Op, Var)
$(f_{var-dep}, Var_1, Var_2)$
…

Feature templates

23 templates

Actual graphs have >1K nodes



Dependency graphs

binary features

f_1	i	n
f_2	p	s
f_3	a	b
f_4	i	i
f_5	i	j
f_6	p	p

3-factor

1	i	i
1	j	i
1	p	p

4-factor

1	i	i	k
1	j	i	a
1	p	p	v

Binary features and factors

Train model

	name1	name2	weight
f_1	i	n	0.4
f_2	p	s	0.5
f_3	a	b	0.2
f_4	i	i	0.3
f_5	i	j	0.6
f_6	p	p	0.4

3-factor			weight
1	i	i	0.4
1	j	i	0.2
1	p	p	0.1

4-factor				weight
1	i	i	k	0.3
1	j	i	a	0.5
1	p	p	v	0.2

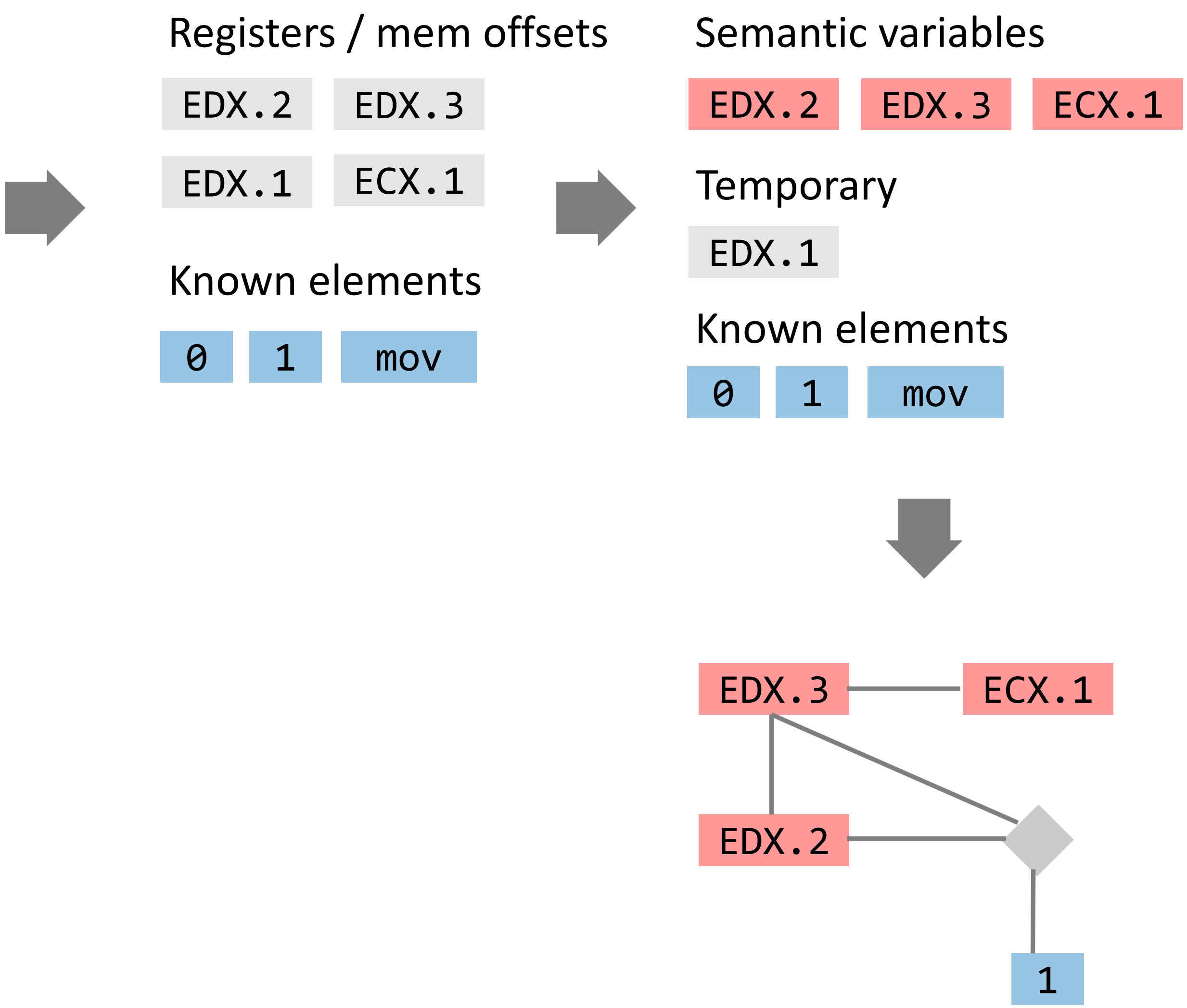
Find ***weights*** that maximize $P(\vec{U} = \vec{u} | \vec{K} = \vec{k_i})$ for all training samples $(\vec{u_i}, \vec{k_i})$

End-to-end recovery of debug information

Recovering debug information

```
<sum> start :
  mov   4(%esp), %ecx
  mov   $0, %eax
  mov   $1, %edx
  add   %edx, %eax
  add   $1, %edx.2
  cmp   %ecx.1, %edx.3
  jne   8048400
  repz ret
<sum> end
```

Stripped binary

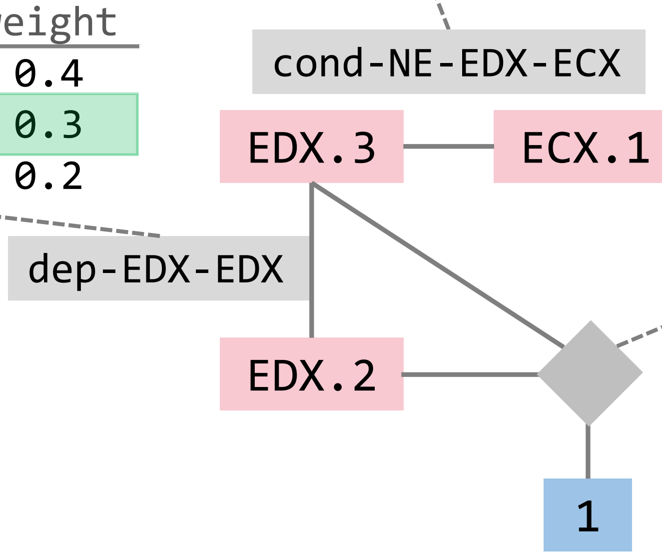


Recovering debug information

MAP inference

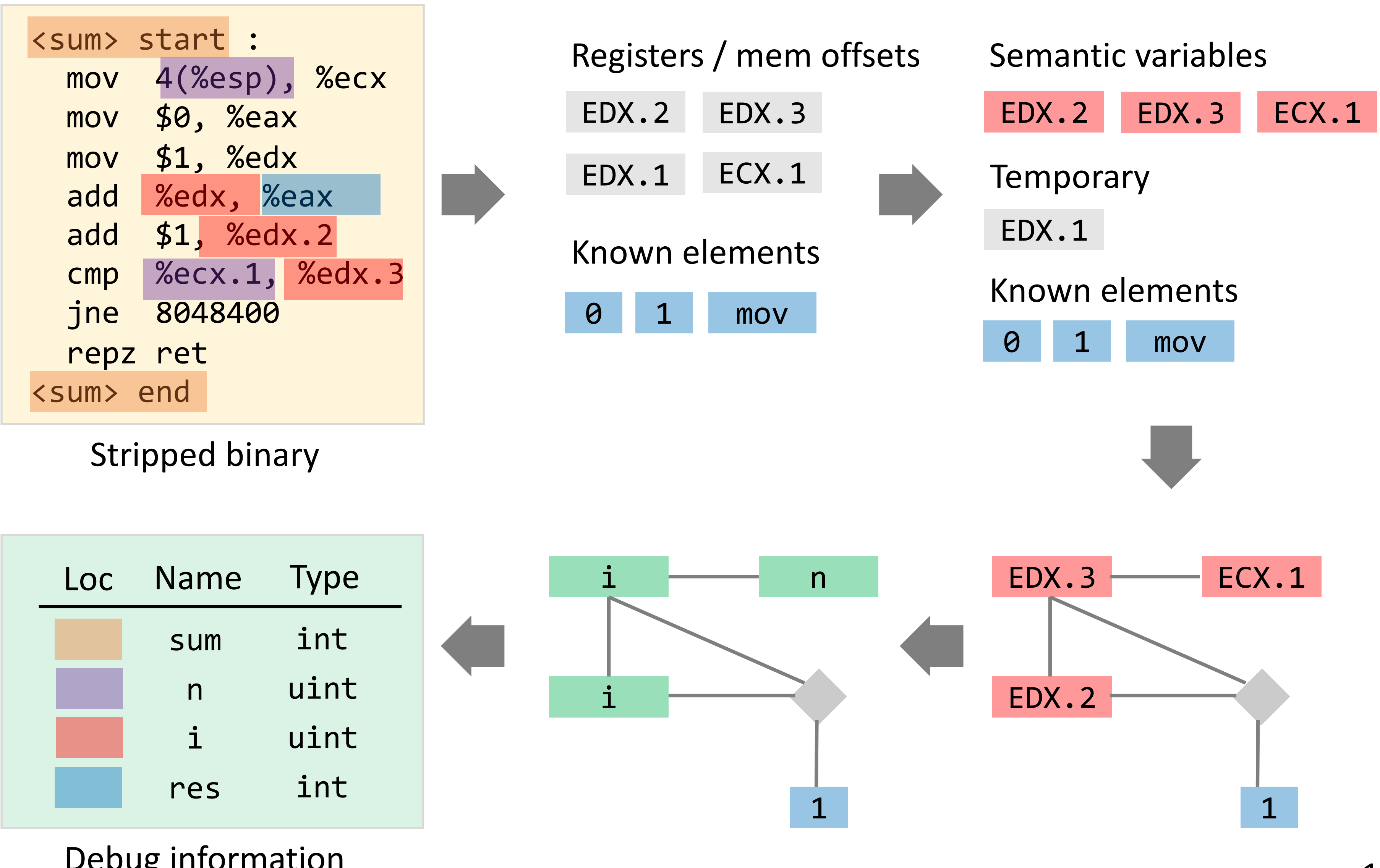
	EDX.3	ECX.1	weight
f_1	i	n	0.5
f_2	p	s	0.3
f_3	a	b	0.1

	EDX.2	EDX.3	weight
f_4	p	p	0.4
f_5	i	i	0.3
f_6	i	j	0.2



1	EDX.2	EDX.3	weight
1	i	i	0.8
1	j	i	0.6
1	p	p	0.3

Recovering debug information



DeBIN implementation

DeBIN implementation

Static analysis: BAP

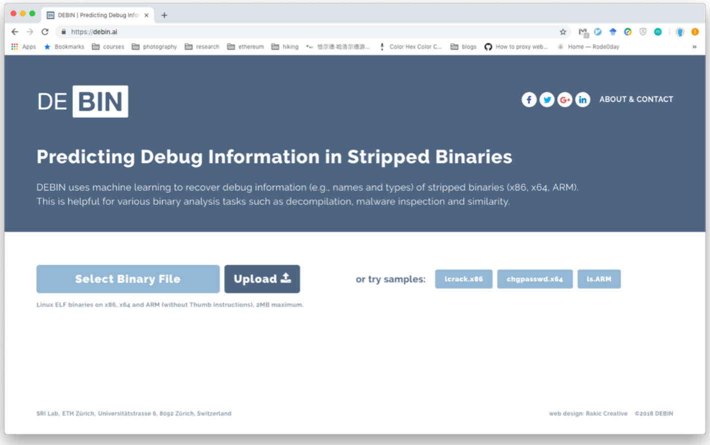
https://github.com/BinaryAnalysisPlatform/bap/

Learning and inference

http://scikit-learn.org

http://nice2predict.org

830 Linux packages
x86, x64, ARM

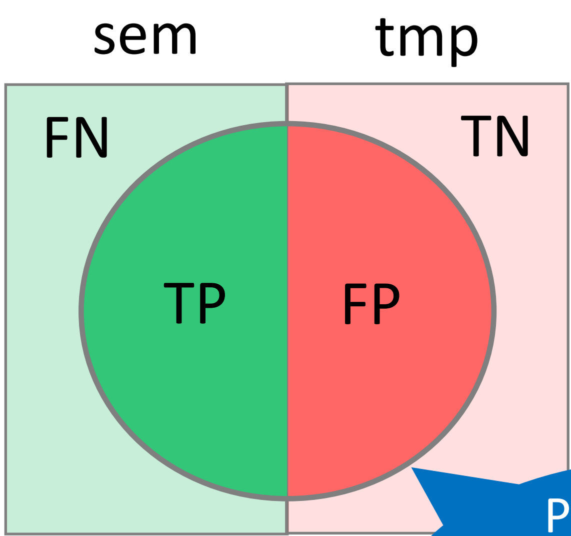


https://debin.ai

DeBIN evaluation

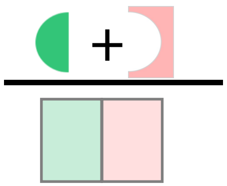
1. How accurate is DeBIN's variable recovery?
2. How accurate is DeBIN's name and type prediction?
3. Is DeBIN useful for malware inspection?

Variable recovery accuracy



Predicted as semantic registers and memory offsets

$$\text{Accuracy} = \frac{|\mathrm{TP}|+|\mathrm{TN}|}{|\mathrm{sem}|+|\mathrm{tmp}|} =$$

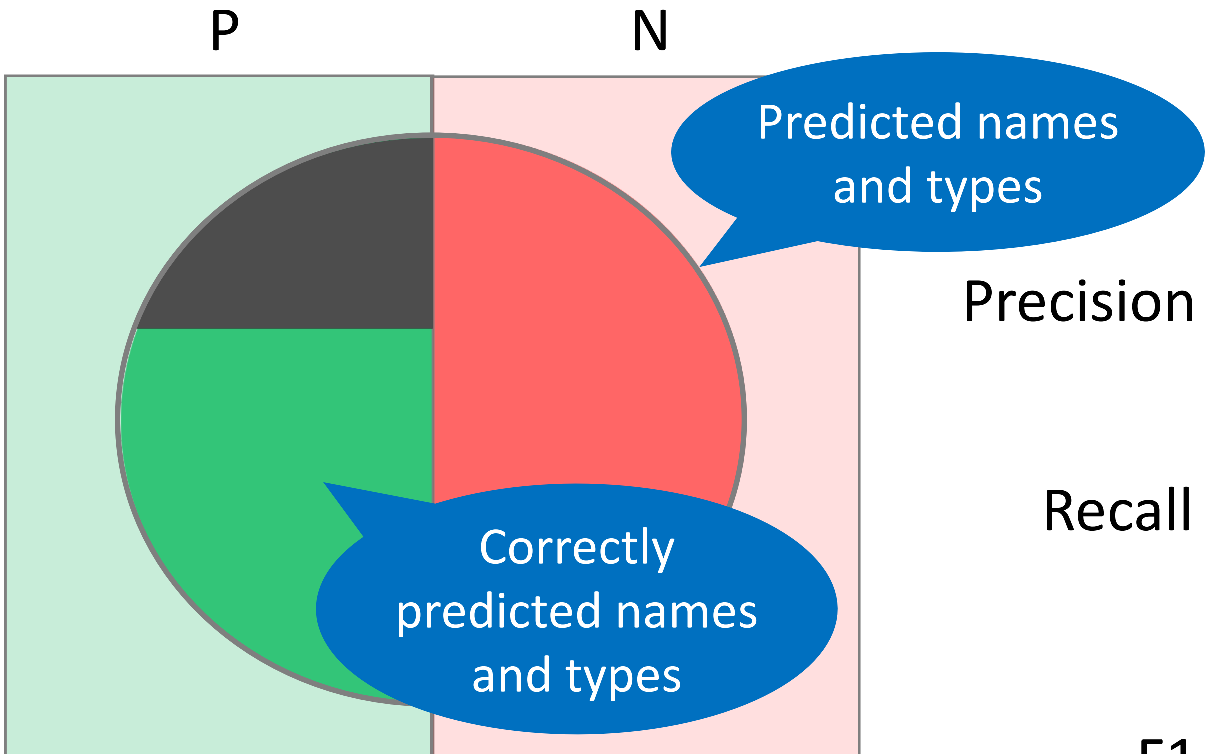


Results

Arch	Accuracy
x86	87.1%
x64	88.9%
ARM	90.6%

DeBIN recovers variables with nearly 90% accuracy

Name and type prediction accuracy



$$\text{Precision} = \frac{|CP|}{|PN|}$$

$$\text{Recall} = \frac{|CP|}{|P|}$$

$$F1 = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Total names and types (P) =

Predicted names and types (PN) =

Correct Predictions (CP) =

Evaluation of name and type prediction

Arch		Precision	Recall	F1
x86	Name	62.6	62.5	62.5
	Type	63.7	63.7	63.7
	Overall	63.1	63.1	63.1
x64	Name	63.5	63.1	63.3
	Type	74.1	73.4	73.8
	Overall	68.8	68.3	68.6
ARM	Name	61.6	61.3	61.5
	Type	66.8	68.0	67.4
	Overall	64.2	64.7	64.5

Consistent precision/recall of roughly 65%

Malware inspection

We inspected 35 x86 malware samples from VirusShare

Manipulating DNS settings

```
int sub_80534BA() {
  ...
  if ( dword_8063320 <= 0 ) {
    v1 = ("/etc/resolv.conf", 'r');
    if (v1 || (v1 =
       sub_8053B1("resolv.conf"))){
       ...
       ...
    }
}
```

DEBIN →

```
int rfc1035_init_resolv() {
  ...
  if ( num_entries <= 0 ) {
    v0 = ("/etc/resolv.conf", 'r');
    if (v0 || (v1 =
       fopen64("resolv.conf"))){
      // code to read and
      // manipulate DNS settings
    }
}
```

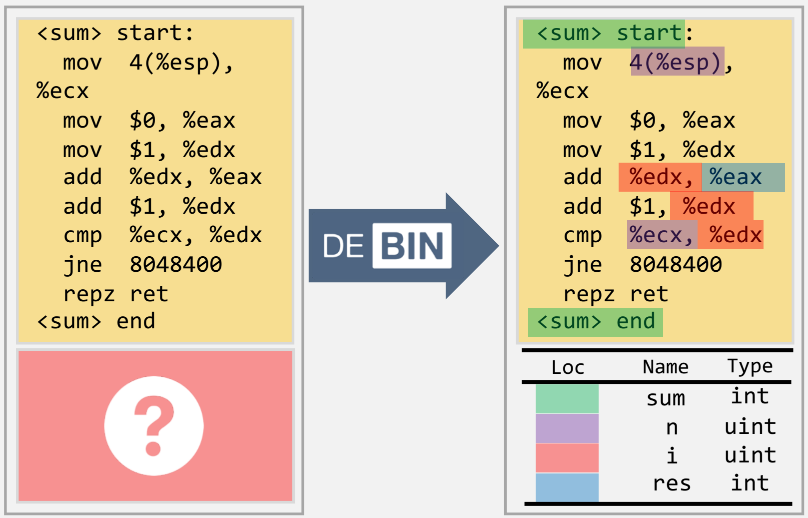
Leakage of sensitive data

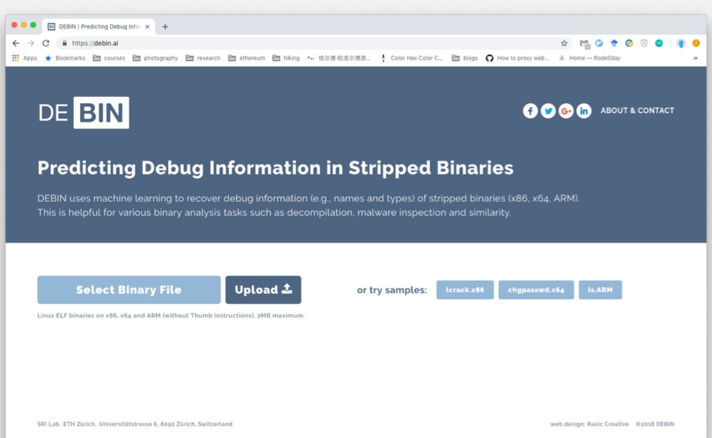
```
If (sub_806d9f0(args) >= 0) {
  ...
  sub_80522B0(args);
  ...
}
```

DEBIN →

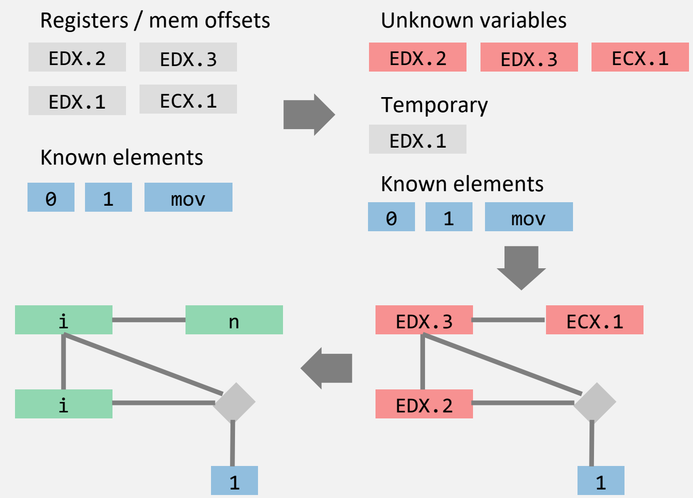
```
If (setsockopt(args) >= 0) {
  ...
  sendto(args);
  ...
}
```

Summary

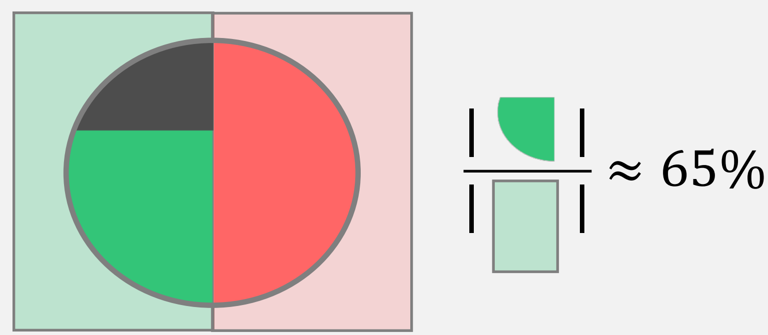




Try online: https://debin.ai



Two-stage prediction process



High precision and accuracy