Frankenstein

Stitching Malware from Benign Binaries

Vishwath Mohan and Kevin W Hamlen

University of Texas at Dallas

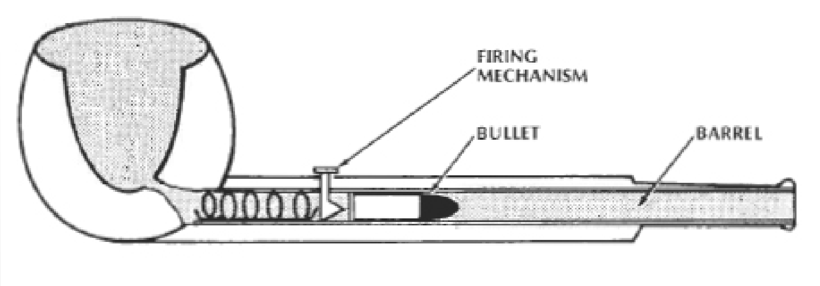
Supported in part by AFOSR

Agenda

- (Drawbacks in) current approaches to obfuscation
- Our approach
- Implementation
- Results

The motivation

- Extend the malware obfuscation arsenal



The motivation

- The current state of detection -
  - Largely static and feature-based
  - Complex analysis only for suspicious files

- The current state of obfuscation -
  - Encryption

  - Virtualization

  - Metamorphism

The motivation

- The current state of detection -
    - Largely static and feature-based
    - Complex analysis only for suspicious files

- The current state of obfuscation -
    - Encryption, or "I'm going to look suspicious"

    - Virtualization

    - Metamorphism

The motivation

- The current state of detection -
  - Largely static and feature-based
  - Complex analysis only for suspicious files
- The current state of obfuscation -
  - Encryption, or "I'm going to look suspicious"
  - Virtualization, or "I'm going to carry a virtual machine around"
  - Metamorphism

The motivation

- The current state of detection -
  - Largely static and feature-based
  - Complex analysis only for suspicious files

- The current state of obfuscation -
  - Encryption, or "I'm going to look suspicious"

  - Virtualization, or "I'm going to carry a virtual machine around"

  - Metamorphism, or "I'm going to change my code each time (but only in simple, detectable ways)"

The motivation

- Gadgets are very interesting
  - Turing-complete functionality
  - Found everywhere
  - Classifiably benign!
- So why not use gadgets to compose your malware

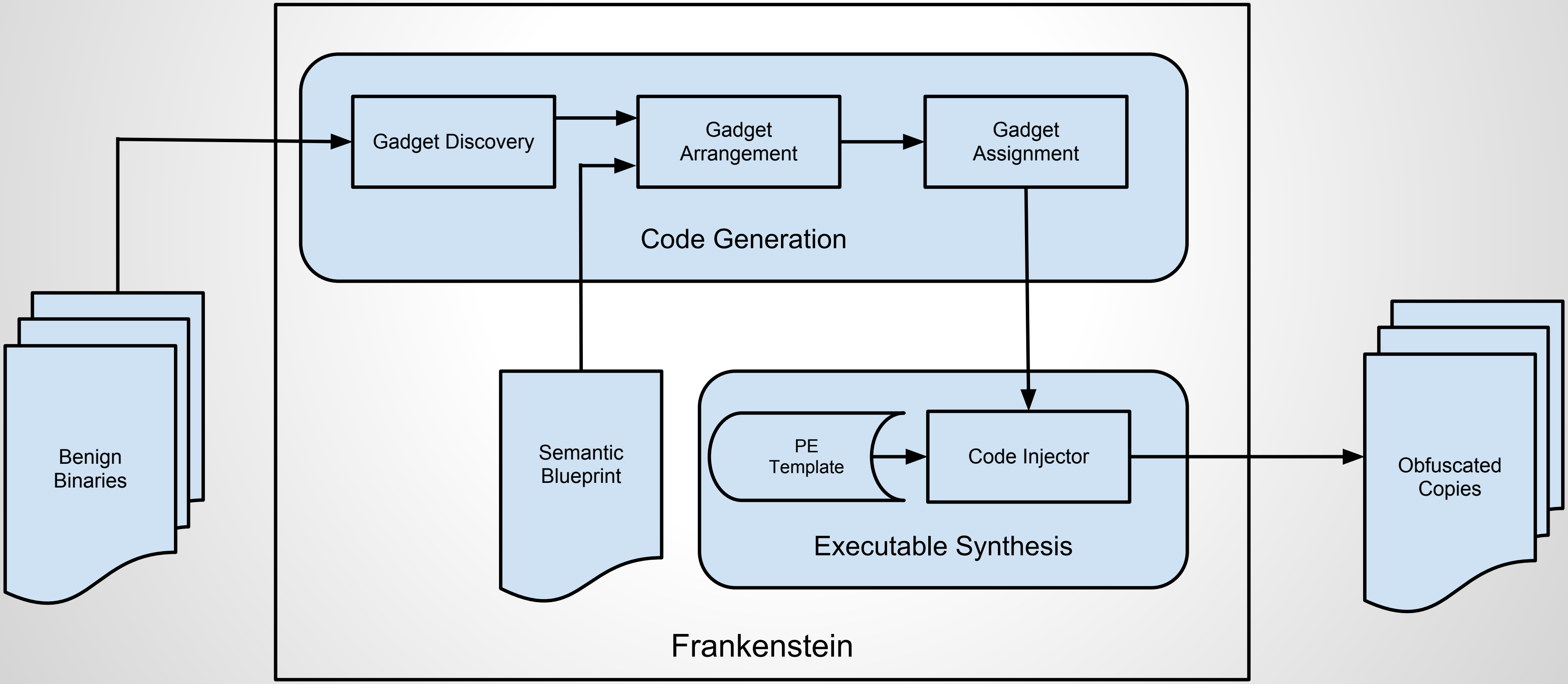
Gadget:
A sequence of bytes representing a valid instruction sequence, and ending in return, that can be used to perform a semantically specific task.

Eg:
  mov eax, ebx
  ret
is a gadget that moves a value from one register (*ebx*) to another (*eax*)

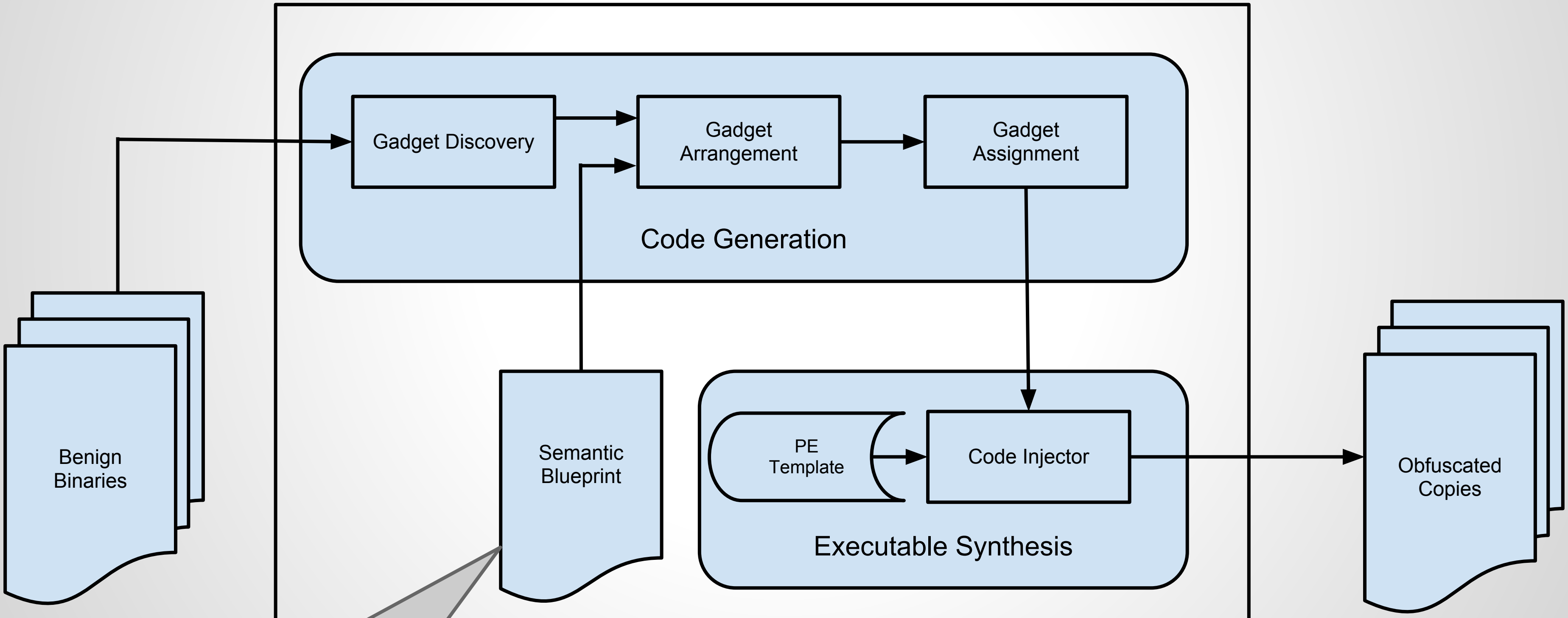
The idea - Or Mary Shelley's recipe

- Consult a high-level blueprint of the ~~human body~~ malware
- Harvest ~~organs~~ gadgets from ~~cadavers~~ benign programs
- Stitch them together to (re)create ~~Frankenstein~~ the malware

Overview


Gadget Discovery
Gadget Arrangement
Gadget Assignment
Code Generation
Benign Binaries
Semantic Blueprint
PE Template
Code Injector
Executable Synthesis
Obfuscated Copies
Frankenstein

Overview



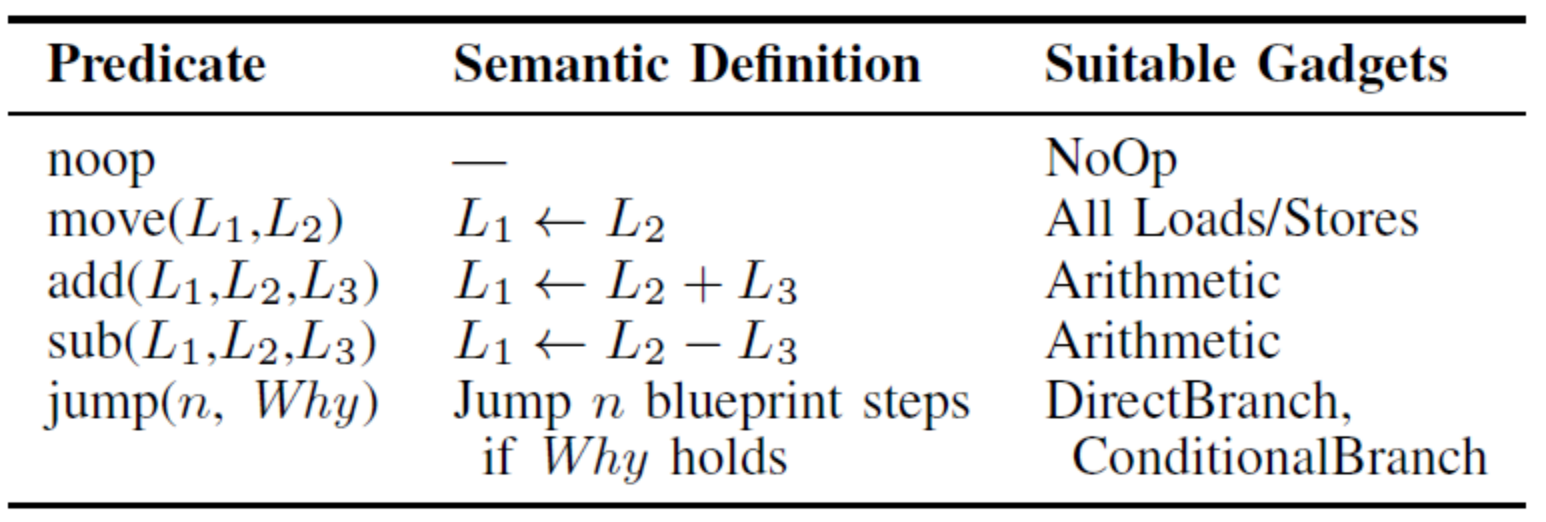
Malware functionality expressed as a set of logical predicates.

Semantic Blueprint

- Eg: semantic blueprint to find the slope of a line given two points (x1,y1) and (x2,y2)

```
evil_slope :=
  sub(L1, y1, y2), // L1 = y1 - y2
  sub(L2, x1, x2), // L2 = x1 - x2
  div(L3, L1, L2).  // L3 = (y1-y2) / (x1-x2)
```

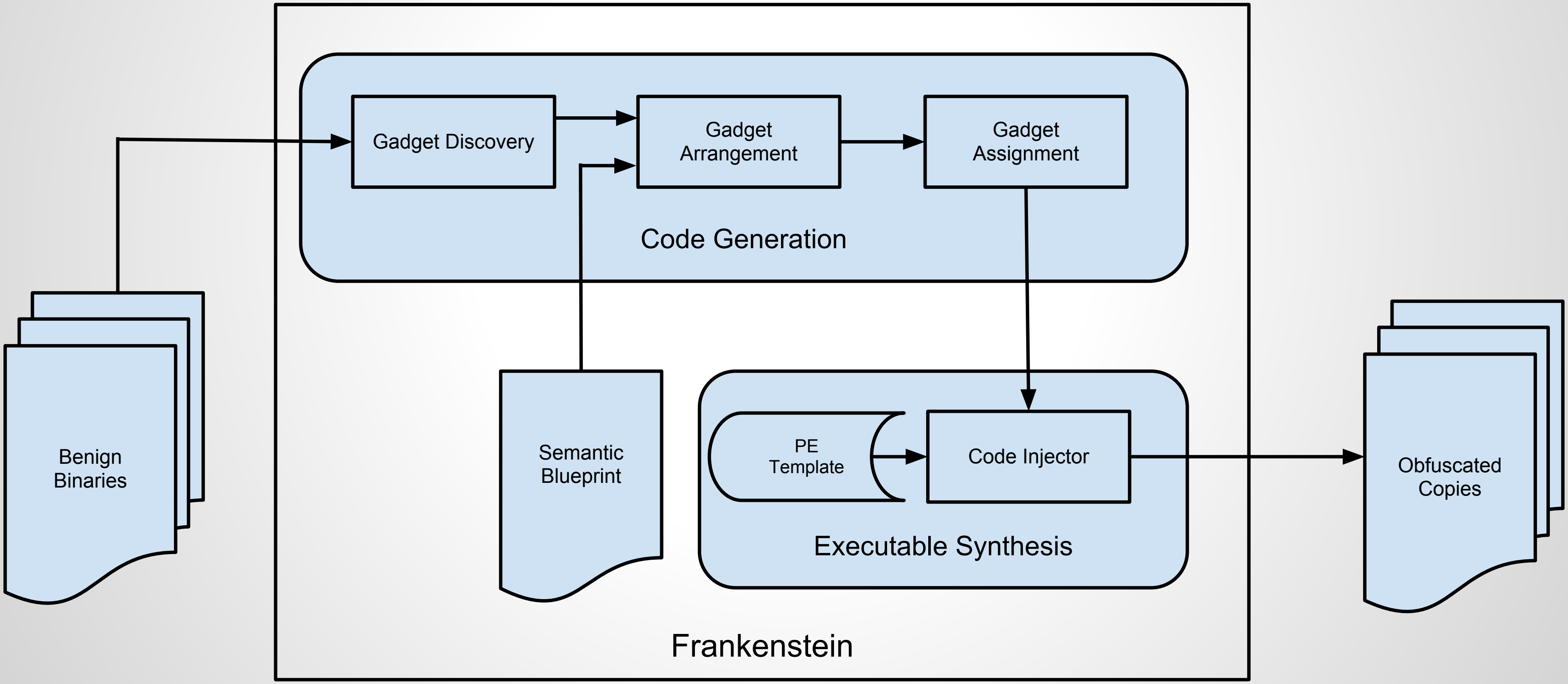
Semantic Blueprint



Predicate	**Semantic Definition**	**Suitable Gadgets**
noop	—	NoOp
move(L_1,L_2)	$L_1 \leftarrow L_2$	All Loads/Stores
add(L_1,L_2,L_3)	$L_1 \leftarrow L_2 + L_3$	Arithmetic
sub(L_1,L_2,L_3)	$L_1 \leftarrow L_2 - L_3$	Arithmetic
jump(n, Why)	Jump n blueprint steps if Why holds	DirectBranch, ConditionalBranch

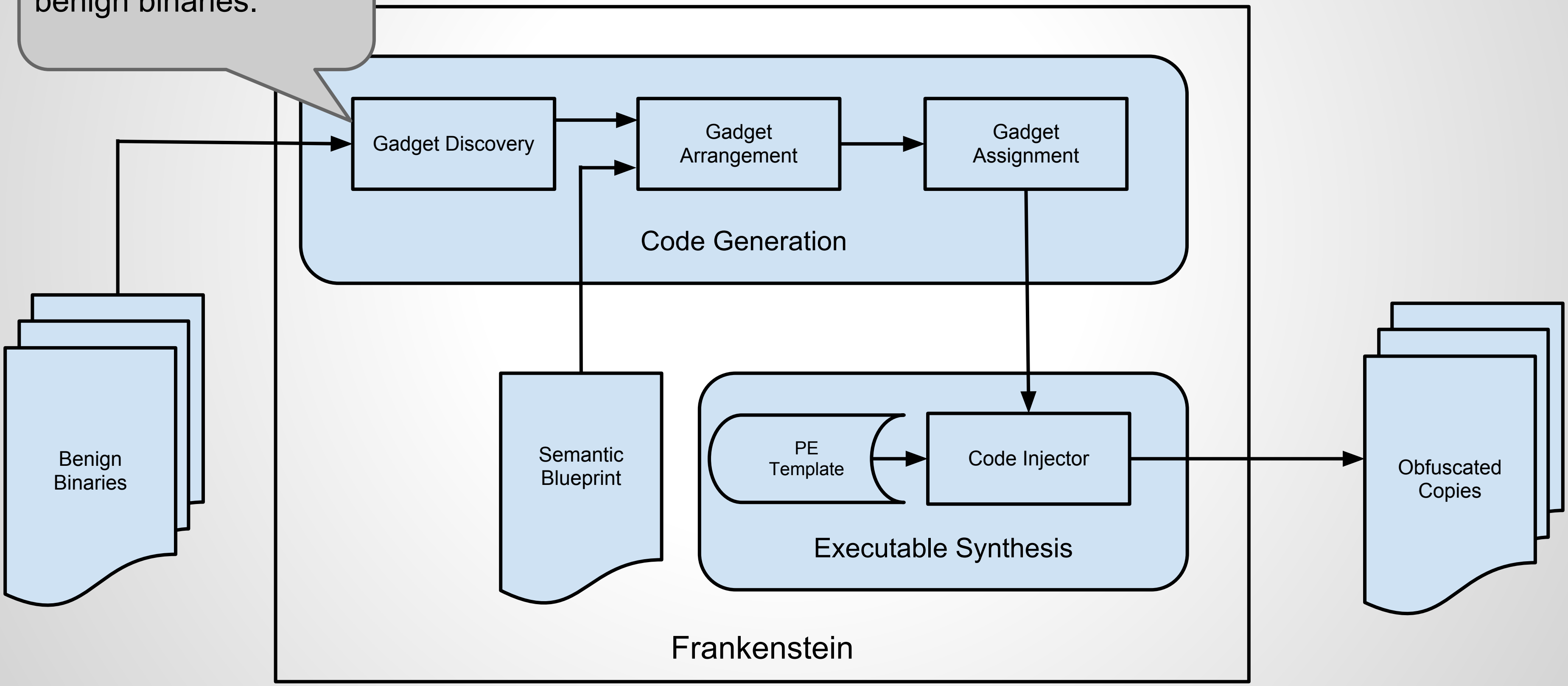
- Logical predicates represent actions
  - Predicates can be layered for abstraction
  - Lowest layer predicates match multiple gadgets
  - The goal is to encode *what* rather than *how*

Overview


Gadget Discovery
Gadget Arrangement
Gadget Assignment
Code Generation
Benign Binaries
Semantic Blueprint
PE Template
Code Injector
Executable Synthesis
Obfuscated Copies
Frankenstein

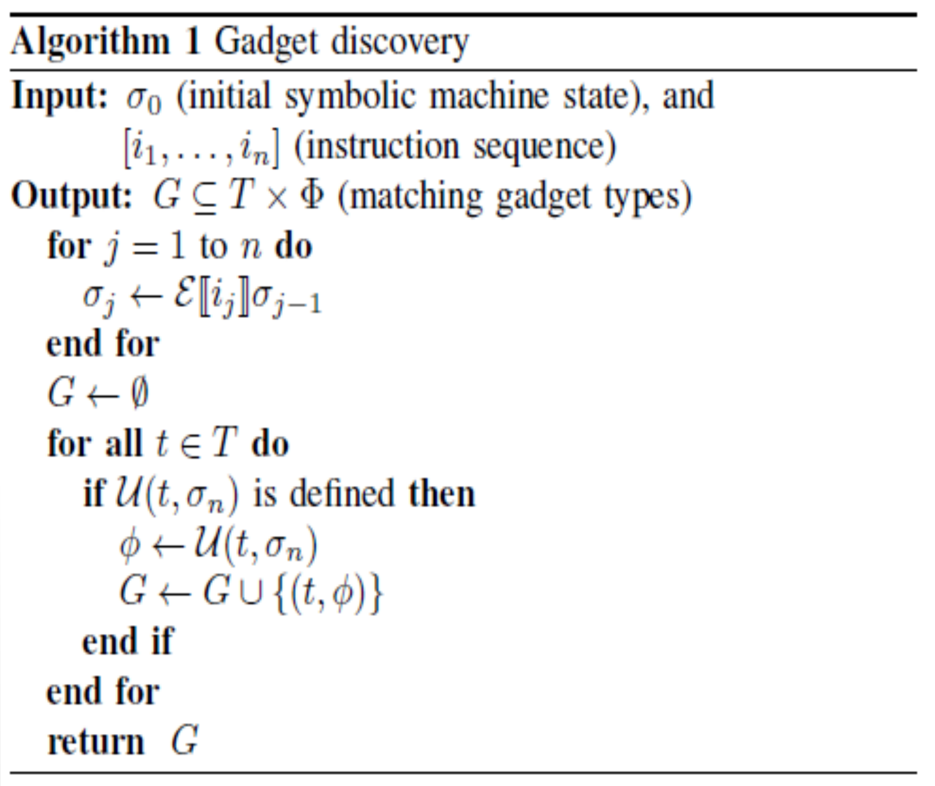
Overview

Builds a database of gadgets from the benign binaries.



Abstract Evaluator

- Input: Instruction sequence, abstract state
- Output: All gadget types that it can match
- Algorithm:



Algorithm 1 Gadget discovery

```
Input: $\sigma_0$ (initial symbolic machine state), and
       $[i_1, \ldots, i_n]$ (instruction sequence)
Output: $G \subseteq T \times \Phi$ (matching gadget types)
  for $j = 1$ to $n$ do
    $\sigma_j \leftarrow \mathcal{E}[\![i_j]\!]\sigma_{j-1}$
  end for
  $G \leftarrow \emptyset$
  for all $t \in T$ do
    if $\mathcal{U}(t, \sigma_n)$ is defined then
      $\phi \leftarrow \mathcal{U}(t, \sigma_n)$
      $G \leftarrow G \cup \{(t, \phi)\}$
    end if
  end for
  return $G$
```

Abstract Evaluator

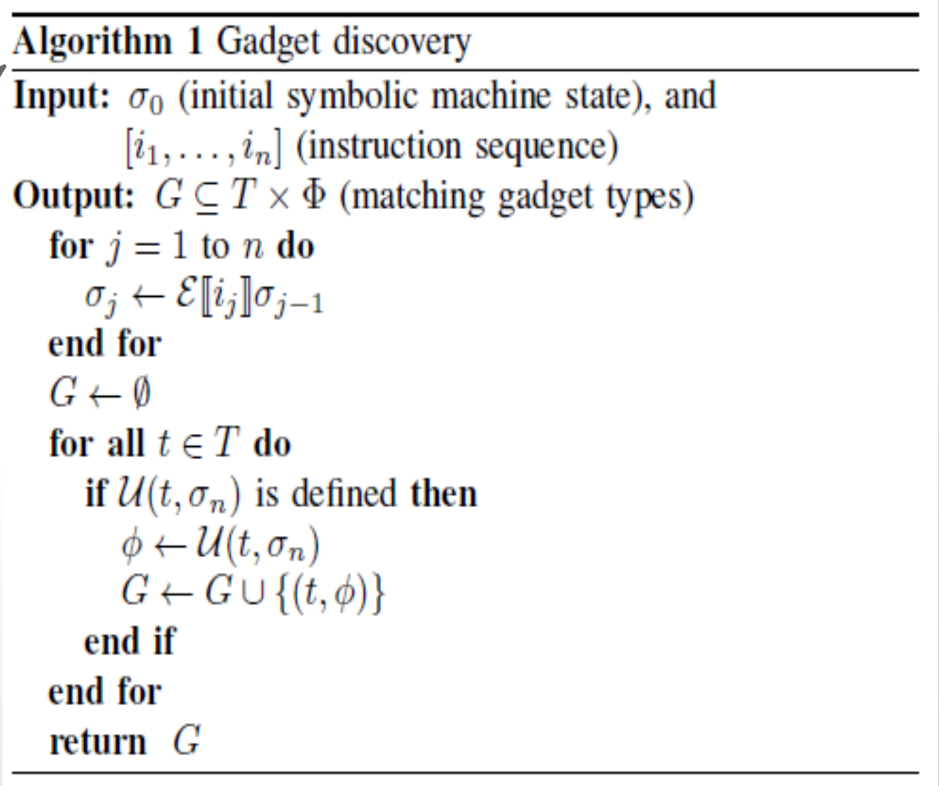
- Input: Instruction sequence, abstract state
- Output: All gadget types that it can match
- Algorithm:

A (contrived) example:
Input:
add eax, ebx
mov ecx, edx

Matched gadget types:
1) Arithmetic(eax **<--** eax + ebx)
clobbers[ecx]

2) LoadReg(ecx **<--** edx)
clobbers[eax]

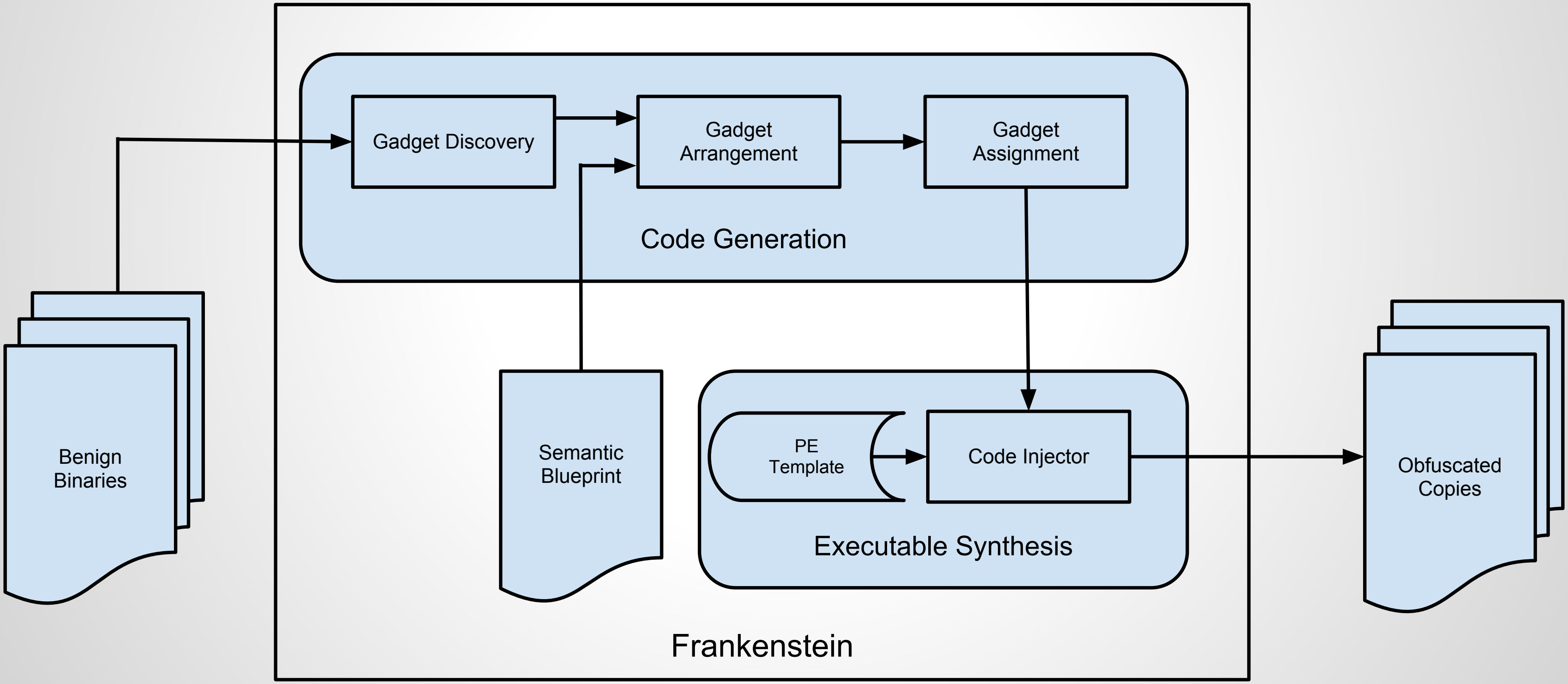
3) Nop
clobbers[eax, ecx]



Algorithm 1 Gadget discovery

```
Input: $\sigma_0$ (initial symbolic machine state), and
       $[i_1, \ldots, i_n]$ (instruction sequence)
Output: $G \subseteq T \times \Phi$ (matching gadget types)
  for $j = 1$ to $n$ do
    $\sigma_j \leftarrow \mathcal{E}[\![i_j]\!]\sigma_{j-1}$
  end for
  $G \leftarrow \emptyset$
  for all $t \in T$ do
    if $\mathcal{U}(t, \sigma_n)$ is defined then
      $\phi \leftarrow \mathcal{U}(t, \sigma_n)$
      $G \leftarrow G \cup \{(t, \phi)\}$
    end if
  end for
  return $G$
```

Overview



Gadget Discovery

Gadget Arrangement

Gadget Assignment

Code Generation

Benign Binaries

Semantic Blueprint

PE Template

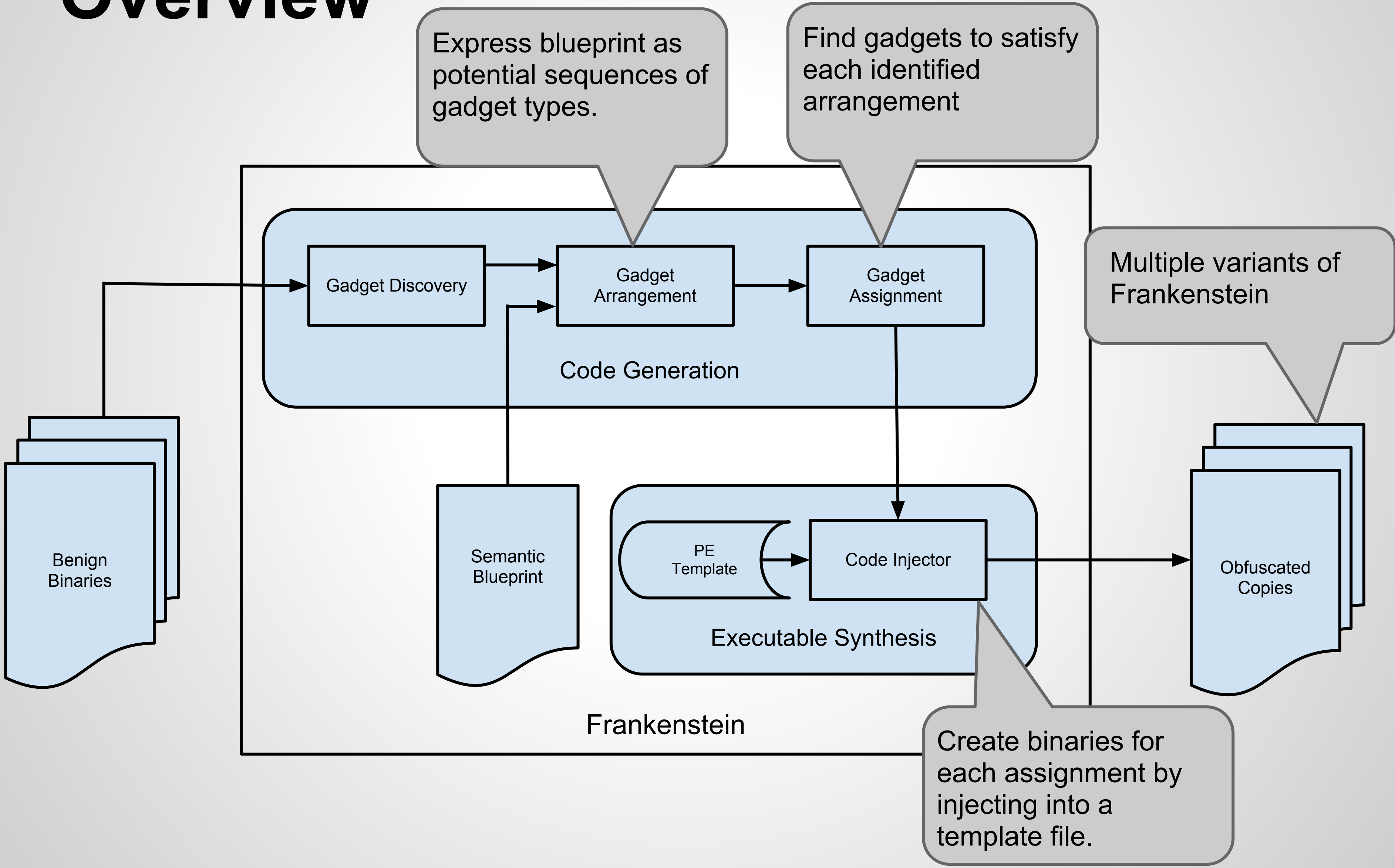
Code Injector

Executable Synthesis

Obfuscated Copies

Frankenstein

Overview



Express blueprint as potential sequences of gadget types.

Find gadgets to satisfy each identified arrangement

Multiple variants of Frankenstein

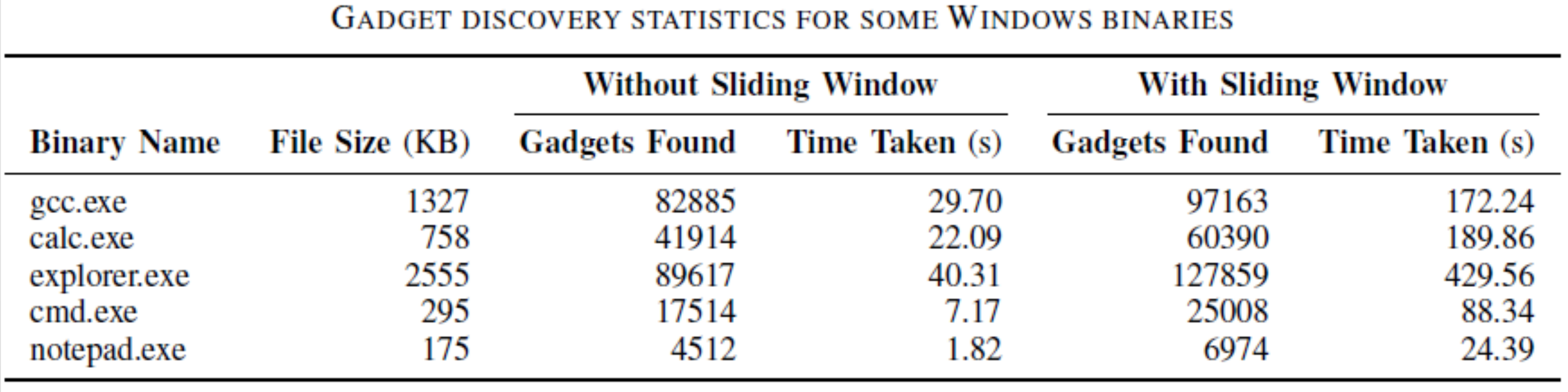
Create binaries for each assignment by injecting into a template file.

The implementation

- Combination of
  - Python - gadget discovery, executable synthesis
  - Prolog - gadget assignment
- Small abstract evaluator
  - Only identifies 8 non-branch instructions
  - Still more than sufficient for good results
- Not self-propagating (yet!)

The results

- Gadget discovery
  - Tested with files from *windows/system32*
  - Limited gadget size to 2-6 instructions



GADGET DISCOVERY STATISTICS FOR SOME WINDOWS BINARIES

		Without Sliding Window		With Sliding Window	
Binary Name	**File Size (KB)**	**Gadgets Found**	**Time Taken (s)**	**Gadgets Found**	**Time Taken (s)**
gcc.exe	1327	82885	29.70	97163	172.24
calc.exe	758	41914	22.09	60390	189.86
explorer.exe	2555	89617	40.31	127859	429.56
cmd.exe	295	17514	7.17	25008	88.34
notepad.exe	175	4512	1.82	6974	24.39

- Verdict:
  - 46 gadgets per KB of code
  - 2300 gadgets per second
  - Encouraging!

The results

- Generated working mutants
  - Gadgets only from explorer.exe
  - Insertion Sort
  - XOR-based olgimorphism
- Found over 10,000 viable gadget assignments each!
- Size increase a little less than double on average

The results

- Compositionally benign?
- Analyzed fresh n-grams across 20 mutants
  - 2% common across 3 or more
  - 0.3% common across 5 or more
  - 0% common across 7 or more
- i.e: No defining n-grams across 35% or more

Conclusion

- New way to obfuscate malware
- Principled approach to metamorphism
- Early results show high potential for non-distinguishability from benign programs
- Definitely worth developing further

Questions?

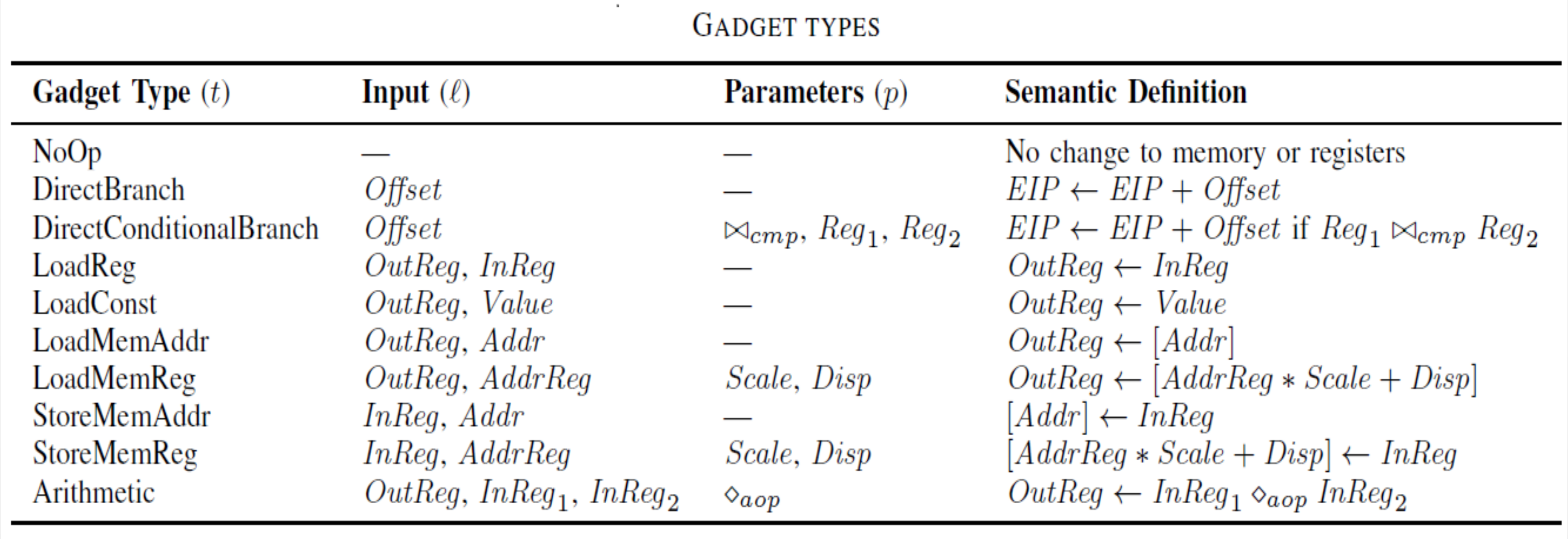
Vishwath Mohan
(vishwath.mohan@gmail.com)

Thank You

Vishwath Mohan
(vishwath.mohan@gmail.com)

Gadgets

- Less constrained notion of a gadget
- More varied (but less than micro-gadgets?)

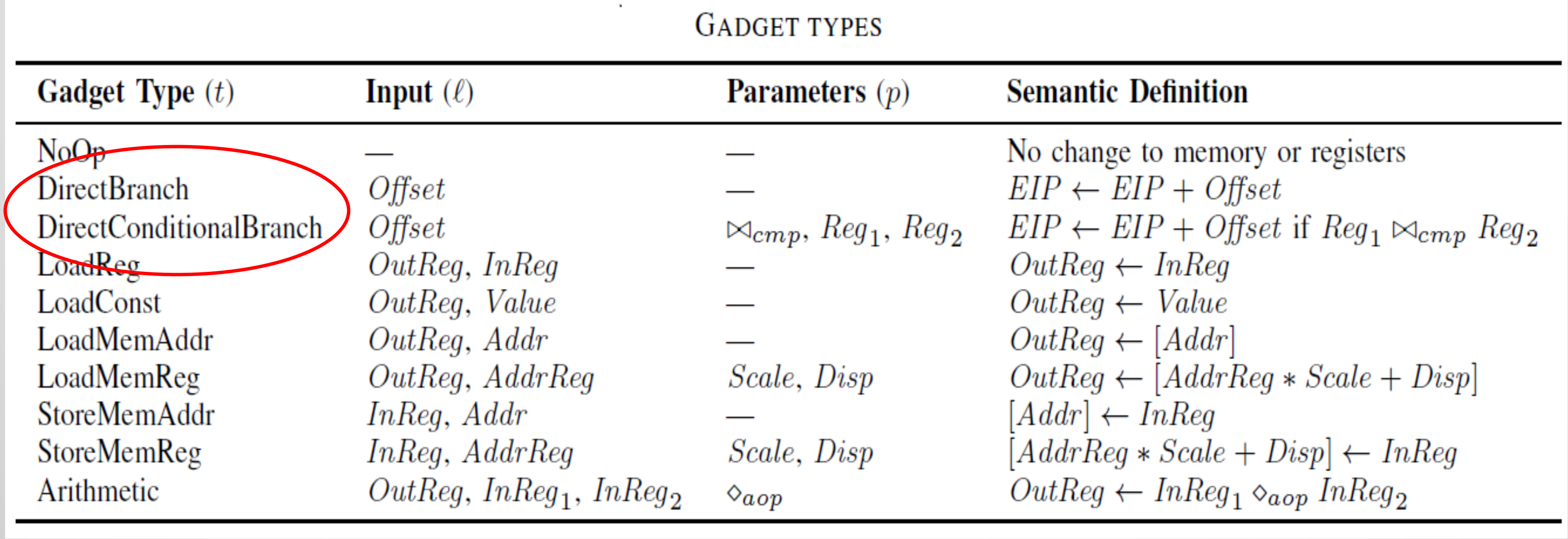


GADGET TYPES

Gadget Type (t)	**Input** (ℓ)	**Parameters** (p)	**Semantic Definition**
NoOp	—	—	No change to memory or registers
DirectBranch	$Offset$	—	$EIP \leftarrow EIP + Offset$
DirectConditionalBranch	$Offset$	$\bowtie_{cmp}, Reg_1, Reg_2$	$EIP \leftarrow EIP + Offset$ if $Reg_1 \bowtie_{cmp} Reg_2$
LoadReg	$OutReg, InReg$	—	$OutReg \leftarrow InReg$
LoadConst	$OutReg, Value$	—	$OutReg \leftarrow Value$
LoadMemAddr	$OutReg, Addr$	—	$OutReg \leftarrow [Addr]$
LoadMemReg	$OutReg, AddrReg$	$Scale, Disp$	$OutReg \leftarrow [AddrReg * Scale + Disp]$
StoreMemAddr	$InReg, Addr$	—	$[Addr] \leftarrow InReg$
StoreMemReg	$InReg, AddrReg$	$Scale, Disp$	$[AddrReg * Scale + Disp] \leftarrow InReg$
Arithmetic	$OutReg, InReg_1, InReg_2$	$\diamond_{aop}$	$OutReg \leftarrow InReg_1 \diamond_{aop} InReg_2$

Gadgets

- Less constrained notion of a gadget
- More varied (but less than micro-gadgets?)



GADGET TYPES

Gadget Type (t)	Input (ℓ)	Parameters (p)	Semantic Definition
NoOp	—	—	No change to memory or registers
DirectBranch	$Offset$	—	$EIP \leftarrow EIP + Offset$
DirectConditionalBranch	$Offset$	$\bowtie_{cmp}, Reg_1, Reg_2$	$EIP \leftarrow EIP + Offset$ if $Reg_1 \bowtie_{cmp} Reg_2$
LoadReg	$OutReg, InReg$	—	$OutReg \leftarrow InReg$
LoadConst	$OutReg, Value$	—	$OutReg \leftarrow Value$
LoadMemAddr	$OutReg, Addr$	—	$OutReg \leftarrow [Addr]$
LoadMemReg	$OutReg, AddrReg$	$Scale, Disp$	$OutReg \leftarrow [AddrReg * Scale + Disp]$
StoreMemAddr	$InReg, Addr$	—	$[Addr] \leftarrow InReg$
StoreMemReg	$InReg, AddrReg$	$Scale, Disp$	$[AddrReg * Scale + Disp] \leftarrow InReg$
Arithmetic	$OutReg, InReg_1, InReg_2$	$\diamond_{aop}$	$OutReg \leftarrow InReg_1 \diamond_{aop} InReg_2$